České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Dryk Jan

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Rozšíření prostředí Process Simulate pro optimalizaci robotických buněk

Pokyny pro vypracování:

V současné době neexistuje rozšíření prostředí Siemens Process Simulate (PS), které by optimalizovalo robotické buňky jako celek. Například rozšíření 'Path Planner' z Process Simulate [1] umožňuje optimalizovat jednotlivé robotické cesty (trajektorie), ale nedokáže zohlednit vazby mezi nimi. Předmětem práce je navrhnout tzv. plugin pro PS, který dokáže upravit parametry modelu robotické buňky tak, aby byla minimalizována doba výrobního cyklu.

Pokyny pro vypracování:

1. Seznamte se s rozhraním PS pro připojení pluginu.
2. Proveďte návrh pluginu a algoritmu pro optimalizaci doby cyklu robotické buňky.
3. Naprogramujte plugin a optimalizační algoritmus.
4. Navrhněte generátor modelů robotických buněk a ten použijte pro otestování pluginu.
5. Plugin otestujte s uživateli.

Seznam odborné literatury:

[1] Siemens, Siemens Product Lifecycle Management Software 2 (IL) Ltd., 2016.
[2] L. Bukata, P. Šůcha, Z. Hanzálek and P. Burget, 'Energy Optimization of Robotic Cells,' in IEEE Transactions on Industrial Informatics, vol. 13, no. 1, pp. 92-102, Feb. 2017.

Vedoucí: Ing. Přemysl Šůcha, Ph.D.

Platnost zadání do konce letního semestru 2017/2018

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

L.S.

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 27.2.2017

Diploma Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

An extension of Process Simulate for optimization of robotic cells

Jan Dryk
Open Informatics

January 8, 2018
Supervisor: Ph.D. Přemysl Šůcha

Acknowledgement / Declaration

First, and foremost I would like to thank my supervisor Ph.D. Přemysl Šůcha for his valuable advice and suggestions while I was writing the thesis as well as for his time.

I would also like to thank Ing. Libor Bukata for his kind help with my research.

Last but not least I would like to thank my colleagues, family and friends for their patience and support during the preparation of this thesis.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

I agree with the utilization of the information presented in my thesis pursuant to Copyright Act 121/2000 Coll., Sec. 60.

Prague, January 8, 2018

.....

Abstrakt / Abstract

Process Simulate od firmy Siemens je průmyslovým standardem v oblasti softwaru pro návrh výrobních linek. Umožňuje výrobcům plánovat a ověřovat montážní linku dlouho než začneme stavět budovu. Tato práce usiluje o to, aby uživatelům pomohla zlepšit kvalitu svých návrhů tím, že jim poskytne rozšíření aplikace zaměřené na optimalizaci. Toto rozšíření poskytuje uživatelské rozhraní k zahrnutému optimalizačnímu algoritmu, jehož cílem je minimalizovat dobu cyklu a zároveň zabránit kolizím. Nejprve robotickou buňku a operaci analyzuje a poté upravuje operaci podle optimálního řešení, které našel optimizační algoritmus. Řešení bylo navrhnuto modulárně, aby mohlo být v budoucnu rozšířeno o sofistikovanější optimalizační algoritmy.

Process Simulate by Siemens is the industry standard in the area of assembly line design software. It allows the manufacturers to plan and validate an assembly line long before breaking the ground. This work strives to help the users improve the quality of their designs by providing them with an optimization plugin. This plugin provides a user interface to the included optimization algorithm which, aims to minimize the cycle time while avoiding any collisions. First, it analyzes the robotic cell and the operation and then it adjusts the operation according to the optimal solution found by the algorithm. The plugin was built with modularity in mind so that it can be extended with more sophisticated optimization algorithms in the future.

Contents

Chapter 1

Introduction

Streamlined manufacturing process always been an important factor for the success of a manufacturer in the market. Quality of the product, however, does not necessarily lead to high profit, which is essential for the company to grow. A product is profitable only if it can be produced with a lesser production cost than market price. While the price is controlled by the market, the cost is easier to positively influence. It can be influenced for example by improving the efficiency of the manufacturing system. Modern manufacturing is highly automated and consists of robotic cells which can produce parts even without an intervention of a human in some cases [1].

When developing a new product the engineers usually create a computer model of the product and based on that they can build prototypes. After the product is refined a different engineering team is tasked with designing an assembly line that could mass produce the parts and assemble them together.

Currently, assembly lines are designed in a specific type of CAD software, which allows the simulation of the full assembly sequence. This way the engineers can validate correctness of the design including human factors, as well as performance indicators like the production cycle time or lead time. One example of such software is the Tecnomatix suite by SIEMENS. Tecnomatix Process Simulate (see Figure 1.1) is an industry leading software for digital manufacturing, used by the likes of Volkswagen or Samsung [2].

When designing an assembly line for a product the focus is on the successful creation of the product, which in itself is no mean feat. Then searching for the optimal layout of the robotic cell and schedule of the tasks that need to be executed for the desired result is an superhuman task. That leaves a lot of potential for computer assistance in this area.

The vision of the future, predicting the 4th industrial revolution (see

Figure 1.1: Process Simulate

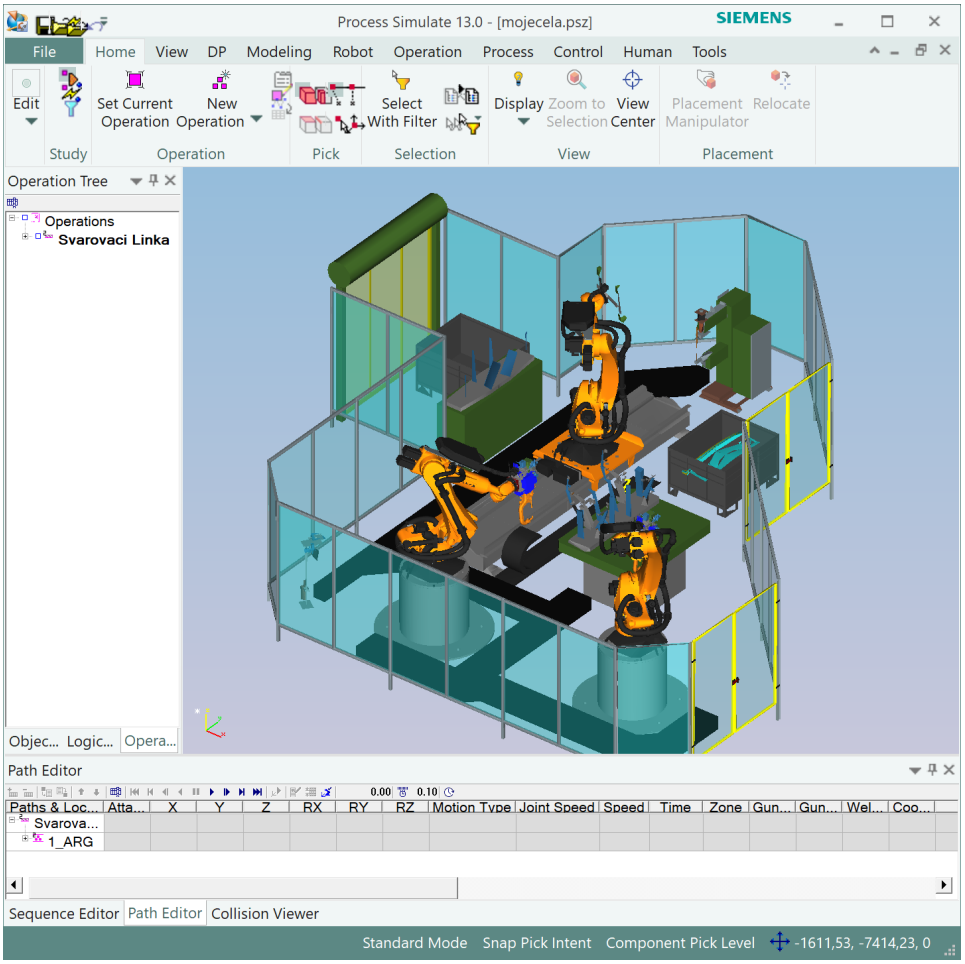
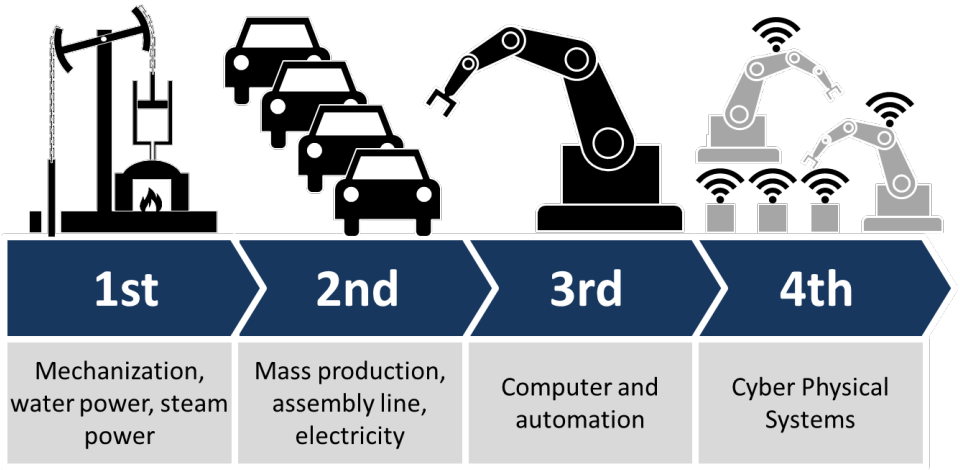


Figure 1.2, was given a name "Industry 4.0". This term was popularized by the German government when they recognized the value of innovation in this field and started supporting the movement. Lasi et al. [3] also outline two different directions Industry 4.0 projects can take. A technological push which consists of further increasing mechanization and automation, digitization and networking, and miniaturization. Or an application pull which has mainly these areas: time to market, individualization on demand, flexibility, decentralization and resource efficiency.

In the context of this work the last point is the most interesting. Resource efficiency is important for several reasons. For example the increase of resource prices, government regulations as well as higher sensitivity to our environment. Due to these, a more intensive focus on sustainability and ef-

Figure 1.2: Industry 4.0, by Christoph Roser at AllAboutLean.com



fectivity of the processes is required. The aim is an economic and ecological increase in efficiency.

1.1 Motivation

As we touched in the previous chapter, optimization of manufacturing processes is a fascinating field with a lot of potential.

Optimizing cycle time allows the manufacturer to produce more units in the same time-span. That, in turn, increases the overall effectivity of the factory and the passive resource usage per unit (lighting, heating, employees, etc.) in contrast to the active resource consumption (robot movement, welding, etc.).

Cycle time, also known as production rate or period, corresponds to a time interval (1/throughput) between two consecutively leaving work-pieces. The duration of the start-up phase called a leading time in cyclic scheduling is the total time required for the first work-piece to be processed by the robotic cell, or in other words, it is a time difference between the time the work-piece entered and the time it left the robotic cell. The cycle time is typically shorter than the duration of the start-up phase, and as a consequence, there is usually more unfinished work-pieces in the robotic cell at once [4].

An alternative to optimizing cycle time is to optimize the energy usage of the robots. Usually, it is possible to improve the energy usage without

increasing the cycle time. Even a small reduction in resource usage could have an enormous impact on the finances of the company. For example in the case of General Motors, their factories drew about nine terawatt hours in the year 2015 [5]. According to Meike et al. [6], about 8% of the energy used in the manufacturing plants is consumed by industrial robots. Assuming consumer pricing of 10 cents per kWh even with 1 percent improvements in energy efficiency this roughly equates to $ 720000 in potential savings. In addition to that, the manufacturing business is also affected by government regulations such as the plan of the European Union for energy savings [7] which strive to reduce the emissions and resource usage.

1.2 Related Work

Theoretical optimization of processes in robotic cells, due to the remarkable improvements, is not revolutionary. Due to its high usage of industrial robots, optimization is often linked to the automotive industry. However, the concepts translate to different kinds of industries. Moreover, all sorts of factors can be optimized. For example, as shown by R. G. Fenton et al. [8], the location of the robots in a robotic cell can have a profound effect on the cycle time. It is possible to obtain the optimal position using a numerical optimization routine and a kinematic computer graphics simulation program.

Another approach to optimize the cycle time of robotic cells was shown by Jiafan Zhang et al. [9]. In their research, the focus was laid mainly on scheduling movement of robots with single or dual grippers. The throughput of most dual-gripper robots can be improved using the method their team has presented in this article.

Moreover, recently Edvin Åblad et al. [10] took a look at the practical challenges of real assembly line designs. Rather than focusing on certain parts of the robot movement, this group chose to tackle, collision resolution, a different factor influencing the cycle time. Collisions, which are another problem relevant to this thesis, are avoided by introducing synchronization schemes among the robots. These synchronization locks are preventing shared volumes of the workspaces to be simultaneously entered, which is a safe way of avoiding issues. On the other hand, it also has a negative impact on the cycle time. Edvin and the team show a new approach to maximizing throughput while eliminating all synchronizations among robots.

With the collaboration with one of the top players in the automotive industry, Davis Meike et al. [6] investigates potential energy savings on

robotic assembly lines for the automotive industry. Davis and the team present two practical methods for reducing the overall energy consumption. The methods entail the implementation of energy-optimal trajectories obtained utilizing time scaling, concerning the robots' motion from the last process point to the home positions and reduction of energy consumption by releasing the actuator brakes earlier when the robots are kept stationary. Notable are also the results which were simulated based on input from a real manufacturing plant. In the future, it's likely that some manufacturers might choose even to sacrifice cycle time to reach higher energy efficiency of the factory.

Building on top of the work of Meike et al., a study by L. Bukata et al. [4] focuses more narrowly on the energy optimization of industrial robotic cells. They have devised a mathematical model, which takes into account various robot speeds, positions, power-saving modes, and alternative orders of operations. Furthermore, a mixed-integer linear programming formulation is included, ready to be used. Due to speed concerns, they also created a hybrid heuristic capable of utilizing multi-core processors. Experiments show that theoretically, the energy consumption can be reduced by as much as 20% merely by optimizing the robot speeds and applying power-saving modes.

Anne-Laure Coiffier [11] wrote a thesis, which also focuses on the Tecnomatix suite of tools. The goal of her work was to find an optimal schedule for a given setup of a robotic cell and a set of operations. In comparison to my work, her approach was to assign tasks to robots, whereas I consider a fixed assignment and manipulate the speed of the robots. Her algorithm performs a mapping of the operations to the given resources, taking into account the material flow. The optimization was conducted by a Depth-First Search algorithm with backtracking rather than ILP, suggesting that a heuristic approach might be worth considering.

1.3 Contribution

Interfacing with a program capable of kinematic simulation, like Process Simulate, can add a lot of value to the optimization process. Certain subtleties of the domain are more straightforward to simulate rather than to capture them in a mathematical model which could make it difficult to mine it from the application and to compute the optimal solution. Moreover, due to the diversity of the methods and objectives, which all lead to an improved manufacturing process, I recognized that the plugin must be modular and flexible enough so that the user can select different optimization cores fo-

cusing on a particular objective.

The main contribution of this thesis is the integration of an optimization algorithm with Process Simulate so that optimization techniques can be brought from the academia to the industry, into the hands of the engineers. The plugin is developed as a foundation stone for future work. As such it is designed to be extendable and reusable. The integration mainly focuses on analyzing the designed robotic cell, setting up a generic optimization process, introducing helpers to get more information from the system and finally adjusting the robotic cell based on results of the optimization.

The work is divided into six chapters. The opening chapter introduces the reader to the industry and defines the goals of the work. The second chapter breaks down the problem at hand and establishes formal notation which is used throughout the rest of the work. After that a MILP model is devised which can be used with a generic solver to provide an optimal solution minimizing cycle time. Then in chapter 4 I introduce Process Simulate and explain in detail its inner workings. The chapter after that focuses on the plugin itself from features to architecture. This chapter is especially important because the plugin is meant to be expandable and reusable for future work. The Second to last chapter is dedicated to a formal validation of the work. The MILP model is benchmarked, and the plugin itself is tested by the users. And finally a conclusion is made with the recommendations for future work.

Chapter 2

Problem Statement

The optimization problem of a robotic cell can be defined as follows. There is a set of robots $R = \{1, \ldots, m\}$ and a graph a graph $G = (V, E, C)$ where its vertices V are operations (welding, moving, painting, waiting ...) and edges represent relations between the operations.

Let $V = \{1, \ldots, n\}$, then every operation $i \in V$ is characteristic by its minimum ($\underline{d}_i$) and maximum($\overline{d}_i$) duration. Additionally, for each robot $r \in R$ I define a set O_r, so that it contains all the operations assigned to it. Each operation is assigned to exactly one robot, and this assignment means that the robot will execute the operation.

Edges (i, j) in the edge set E represent the precedences between operations. Like vertices, an edge (i, j) also has several properties of its own. Mainly, we recognize three different types of edges defined in $type_{i,j}$. A *robot loop* precedence which stems from the sequential order of the operations in the robots schedule, a *link* which sets precedences between two operations of different robots and last but not least a *robot loop reset* which starts the next cycle. Another property of an edge is the delay $D_{i,j} \geq 0$ which is specifying that the following operation v_{to} can start no earlier than $D_{i,j}$ seconds after i has completed in the same cycle.

Finally, there is the collision set C which contains the pairs of operations $(i, j) \in V^2 : i \neq j$ that can't be executed at the same time, as this would result in a collision.

The goal of the optimization algorithm included in this work is to minimize cycle time ω as defined it in Chapter 1.1.

For example the robotic cell in Figure 2.1 has only 2 robots and consists of operations $V = v_1, \ldots, v_6$, with robots having 3 operations each. Robot r_1

has operations v_1 (move around left corner), v_2 (weld point 1), v_3 (weld point 2). Naturally these three operations need to be performed in a sequence, which is ensured by two *robot loop* edges which are marked red. When the robot finishes with operation v_3 he can start working on the next product coming on the assembly line hence the green *robot loop reset* edge from v_3 to v_1. Likewise for robot r_2. Figure 2.1 also shows a *link* edge which is highlighted blue and shows that v_2 must be finished before v_5 starts. Let's say that the areas where v_3 and v_6 operate overlap (the points that are being spot-welded are too close together) and they can't be processed at the same time. A new collision $c \in C$ would be introduced in the graph so that $c = (v_3, v_6)$.

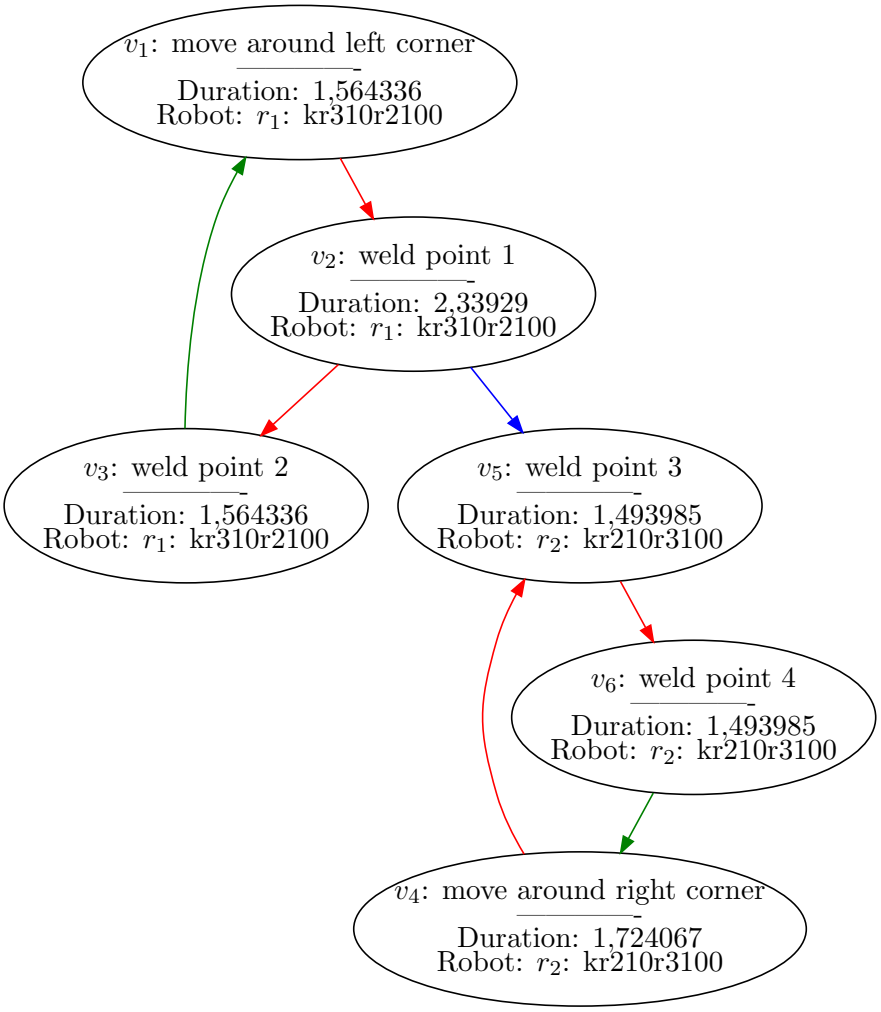


Figure 2.1: Graph generated from a simple robotic cell

Chapter 3

MILP Model

The input to the algorithm is a graph $G = (V, E, C)$ where V is a collection of vertices, E is a collection of edges, and C is a set of collision pairs of vertices. These pairs could cause collisions if executed simultaneously, as described in the previous chapter. For the purposes of the MILP model, we define the following variables. We want to minimize cycle time ω. For operation $i \in V$ we define s_i as the start of the operation. Furthermore, s'_i is a start time of the same operation with respect to cycle time ω. The relation between s_i and s'_i is given by Equation 3.3 where q_i is the index of the execution period.

Equation 3.4 specifies precedences of the operations. The duration is split into the proposed duration d_i, and the wait time d_i^w. The wait time specifies how long the robot waits after the operation is finished before starting to work on the next operation. The duration d_i is constrained by its lower bound $\underline{d}_i$ as well as its upper bound $\overline{d}_i$. In this equation $h_{ij} = 1$ if the edge is a *robot loop reset*, $h_{ij} = 0$ otherwise.

Equation 3.5 specifies the collision constrains. In this model, if there is a collision between operation A and operation B, either A ends before B starts or B ends before A starts, but they can't be running simultaneously.

This is a cyclic scheduling problem with resource constraints, which in our case are the collision zones. This problem is NP-Hard, as shown by Hanen and Munier [12].

$$\min_{\omega} \ \omega \tag{3.1}$$

$$\text{s.t.} \tag{3.2}$$

$$s_i = s'_i + q_i\omega \qquad \forall i \in V \tag{3.3}$$

$$s_i + d_i + d_i^w = s_j + h_{ij}\omega \qquad \forall i,j \in E \tag{3.4}$$

$$s'_i + d_i \leq s'_j + x_{ij}\omega \qquad \forall i,j \in C \tag{3.5}$$

$$x_{ij} + x_{ji} = 1 \qquad \forall i,j \in C \tag{3.6}$$

$$\underline{d}_i \leq d_i \leq \overline{d}_i \qquad \forall i \in V \tag{3.7}$$

$$\text{where:} \tag{3.8}$$

$$\omega \in \mathbb{R}^+ \tag{3.9}$$

$$s_i, s'_i, d_i \in \mathbb{R}^+ \tag{3.10}$$

$$q_i \in \mathbb{Z}^+ \tag{3.11}$$

$$x_{ij}, h_{ij} \in \{1,0\} \tag{3.12}$$

This model has many problems, mainly the multiplication of x_{ij} and ω, therefore we use substitution to remove this multiplication of two variables and simplify the model. In the next step the following substitution was applied: $\tau = \frac{1}{\omega}$.

$$\max_{\tau} \ \tau \tag{3.13}$$

$$\text{s.t.} \tag{3.14}$$

$$s_i\tau = s'_i\tau + q_i \qquad \forall i \in V \tag{3.15}$$

$$s_i\tau + d_i\tau + d_i^w\tau = s_j\tau + h_{ij} \qquad \forall i,j \in E \tag{3.16}$$

$$s'_i\tau + d_i\tau \leq s'_j\tau + x_{ij} \qquad \forall i,j \in C \tag{3.17}$$

$$x_{ij} + x_{ji} = 1 \qquad \forall i,j \in C \tag{3.18}$$

$$\underline{d}_i\tau \leq d_i\tau \leq \overline{d}_i\tau \qquad \forall i \in V \tag{3.19}$$

$$\text{where:} \tag{3.20}$$

$$\tau \in \mathbb{R}^+ \tag{3.21}$$

$$s_i, s'_i, d_i \in \mathbb{R}^+ \tag{3.22}$$

$$\tau \in \mathbb{R}^+ \tag{3.23}$$

$$q_i \in \mathbb{Z}^+ \tag{3.24}$$

$$x_{ij}, h_{ij} \in \{1,0\} \tag{3.25}$$

And finally, after the last substitution $S_i = s_i\tau$, $S'_i = s'_i\tau$, $D_i = d_i\tau$, $D_i^w = d_i^w\tau$ the following is what is implemented in the plug-in.

$$\max_{\tau} \ \tau \tag{3.26}$$

$$\text{s.t.} \tag{3.27}$$

$$S_i = S'_i + q_i \qquad \forall i \in V \tag{3.28}$$

$$S_i + D_i + D_i^w = S_j + h_{ij} \qquad \forall i, j \in E \tag{3.29}$$

$$S'_i + D_i \leq S'_j + x_{ij} \qquad \forall i, j \in C \tag{3.30}$$

$$x_{ij} + x_{ji} = 1 \qquad \forall i, j \in C \tag{3.31}$$

$$\underline{d}_i \tau \leq D_i \leq \overline{d}_i \tau \qquad \forall i \in V \tag{3.32}$$

$$\text{where:} \tag{3.33}$$

$$S_i, S'_i, D_i \in \mathbb{R}^+ \tag{3.34}$$

$$\tau \in \mathbb{R}^+ \tag{3.35}$$

$$q_i \in \mathbb{Z}^+ \tag{3.36}$$

$$x_{ij}, h_{ij} \in \{1, 0\} \tag{3.37}$$

At this stage, we have a linear MILP model which can be solved by most modern MILP solvers. In this final form, I used the MILP model in the optimization algorithm used in the plugin.

Chapter 4

Interface

This chapter I'd like to acquaint the reader with Process Simulate, its inner workings and the programming interface (API) which I used to develop this solution. The reason I present this chapter is to familiarize the users with the terminology and internal processes used in the next chapter (5). First I will explain the tool from the users perspective which should help the user better picture what we try to achieve and how. Then I will describe the most relevant aspects of the API. I will also go through how to accomplish the most common programming exercise, writing Process Simulate plugins.

4.1 Users Perspective

The usefulness of the Process Simulate application stems from its ability to verify the feasibility of an assembly process before breaking ground. Validating reachability and collision clearance is done by simulating the full assembly sequence of the product and the required instruments [13]. Figure 1.1 shows a screenshot of the application.

To walk the readers through Process Simulate, I put together this rather practical text. It includes helpful tips and instructions should the reader want to follow along. While the Process Simulate tool isn't widely available, SIEMENS offers a similar application, RobotExpert, with limited functionality, to which this text is applicable as well.

4.1.1 Create a Study

To begin working with Process Simulate one first has to set up his workspace. The content of every project is divided into two parts. A library and a study. The library contains the models and specifics of the robots, tools, and others, while the study includes information about a specific space, a robotic

cell perhaps, with instances of the models positioned within.

The first thing to create a project is to set up both your library and create a new study. The library is set up at install time and will be shared for all of your studies. Process Simulate will walk you through creating a new study.

I advise after creating a study to turn on floor rendering by selecting *View >Screen Layout >Display Floor* in the ribbon menu.

4.1.2 Inserting Components

You can insert a component by selecting *Modeling >Components >Insert Component* in the ribbon menu. You will be prompted to select a folder containing your model. The supported folders have a name ending with *.co* or *.cojt* which signifies that they're in the proper format.

When inserting a component for the first time, you may receive an error saying that you needed to define its type. The fastest way to define a component's type is to enter the search commands and objects popup (Ctrl+F) and search for a *Define Component Type* command. You will be prompted to select a folder (the same folder as before) and then to choose the type of the component. Types offered include a robot, gun, container, etc. The component will be inserted at the origin point in the study.

4.1.3 Modeling

Process Simulate offers a modest kit of modeling tools. Most of the tools are located in the *Modeling* tab. Before you can start creating geometries you need to select a modeling scope. Modeling scope is a group to which the created geometries will belong. You can select a modeling scope by selecting the component and pressing *Modeling >Scope >Set Modeling Scope*. You can have more than one component in a modeling scope, however, its recommended to end the modeling scope once you've finished altering it. You can end a modeling scope similarly to starting it by pressing the *Modeling >Scope >End Modeling* button.

Like any other CAD software, Process Simulate offers a suite of fundamental 3D modeling tools. In the *Components* group next to the familiar *Insert Component* button, we can find commands for creating brand new components and resources.

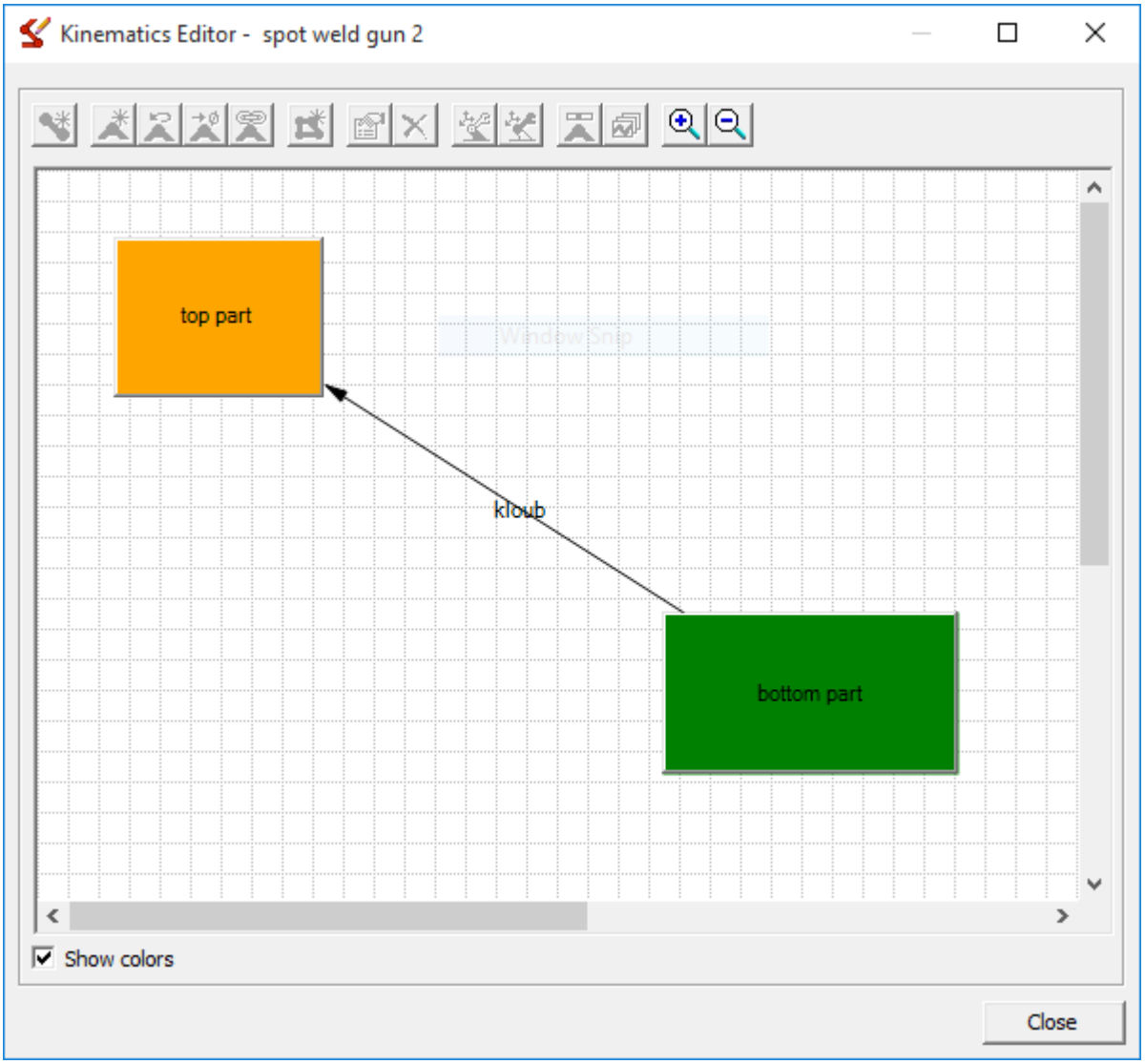
In the next tab named *Layout,* is mainly dedicated to tools for positioning the elements of the study. Notably, the placement manipulator dialog can be accessed using a keyboard shortcut *Alt+P*. This group also contains the *Create Frame* command for which there are several options how to specify a Frame. Frames are oriented points that specify a separate coordinate system within the study, and they are crucial for defining, for example, where do robots hold their tools and so on.

Finally, the *Geometry* group contains commands to create geometries and unify/subtract them together.

4.1.4 Defining Kinematics

Definition of kinematics is the process of defining parts of the model and linking them together with movable joints. It is done using the *Kinematics Editor*, which you can see in Figure 4.1. This feature can be accessed using the *Modeling >Kinematics >Kinematics Editor* command.

Figure 4.1: Kinematics Editor



In the Kinematics Editor, the first button (*Create Link*) will allow you to select all the geometries that belong to a single part. Once there are multiple parts defined a joint can be established by dragging a mouse from a source part onto a destination part. This order is significant in the definition of the joint. The source part will stay stationary while the destination part, to which the arrow is pointed, will be the one moving. As a next step, it is necessary to define the axis of the movement and its limitations. Once defined the joint can be tested in the *Joint Jog* dialog.

4.1.4.1 Poses

A component or a resource can have several predefined poses. A pose is just an assignment of values for each joint which controls how rotated the joint is. Poses can be specified using the *Pose Editor* dialog that has a button in the *Kinematics* group. You can see the *Pose Editor* in Figure 4.2. Figure 4.3 shows how to define an new pose.

Figure 4.2: Pose Editor

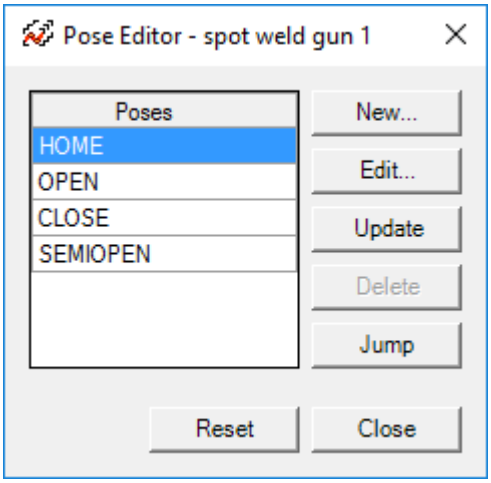
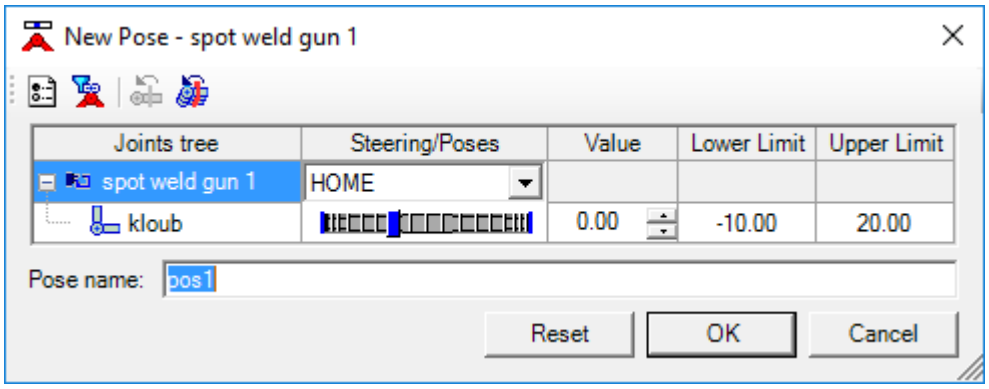


Figure 4.3: New Pose



4.1.5 Robot Tools

We can define a tool for a robot as a component. Each tool type has a set of specific conventions that need to be followed for the application to know

how to work with this tool.

Each gun type tool needs at least two frames. A mounting frame and an effector frame. These can have arbitrary names. However, we will have to configure the robot to know which frame to use as an effector and which as a mounting frame. The *Mount frame* function specifies where and at what angle will the tool be connected to the robot. Effector frame specifies where and what angle should the tool touch the product.

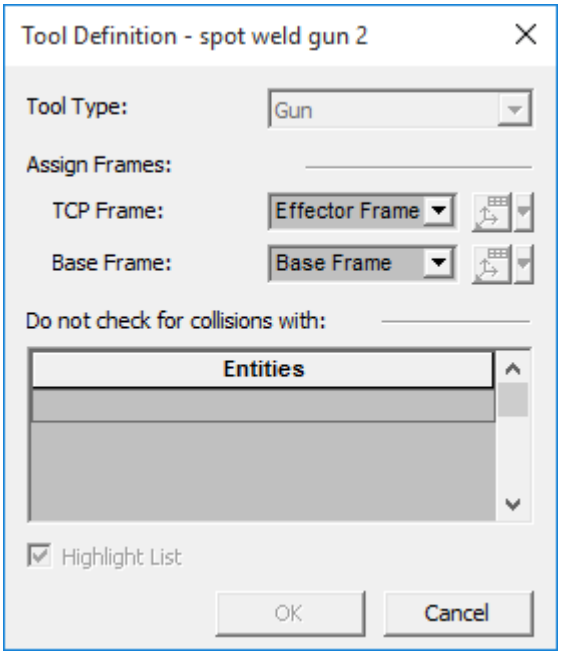
4.1.5.1 Spot Welding Tool

Each tool type has a few different quirks of its own. The specific part about a spot welding gun is that it needs to have defined three poses to help the application generate a clamping animation. These poses need to be named exactly *HOME*, *OPEN*, *SEMIOPEN* and *CLOSE*. Even though it doesn't fit grammatically *CLOSE* is correct without the N at the end, and doesn't work otherwise.

4.1.5.2 Tool Definition

Another step in creating tools is to define it as a tool. We have already marked the component as a *Gun*, *Gripper* or another tool type, but we still need to assign a TCP. TCP stands for Tool Center Point, and it is a frame where the tool affects the product. This dialog, shown in Figure 4.4, can be accessed using the *Modeling* > *Kinematics* > *Tool Definition* command.

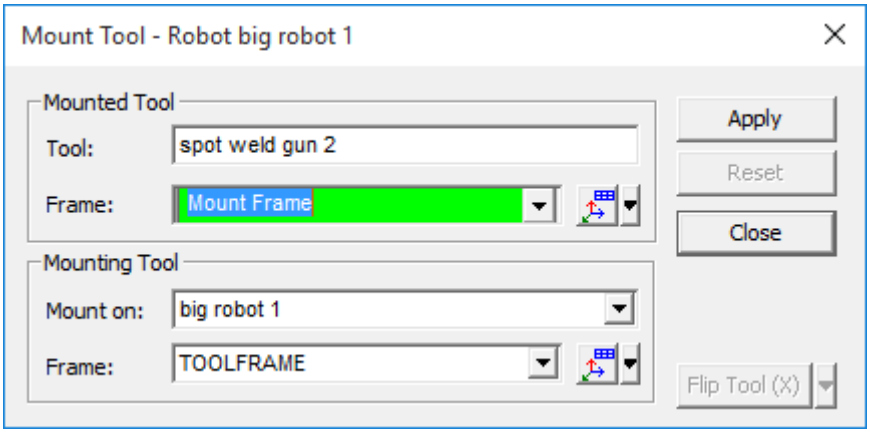
Figure 4.4: Tool Definition



4.1.5.3 Mounting

Now the tools are ready to be mounted. We can do so using the mount dialog which can be accessed from the context menu (*Right Mouse Button*) on the specific robot we want to mount the tool on and select *Mount Tool*. A dialog will be presented, as shown in Figure 4.5. Here, we need to specify what tool we need to mount, using which frame, on which robot and on which frame of the robot respectively. Sometimes not all of the frames owned by the tool are displayed under the combo box. If this is the case enter modeling scope of the tool using the *Set Modeling Scope* command and all the frames should now appear.

Figure 4.5: Mount Tool

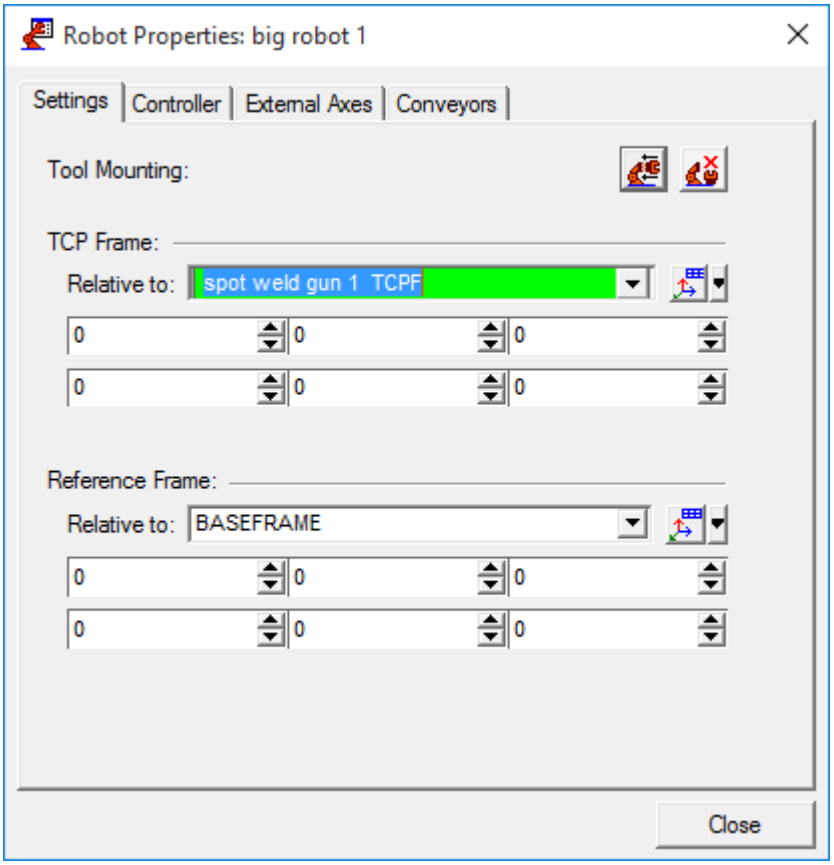


The second the two fields are related to the robot and should be pre-populated with the correct information. If for some reason the robot wasn't well defined and doesn't contain a default tool frame you can set it here.

4.1.5.4 Robot effector frame

Last but not least we need to make sure the robot knows which frame to use to alter the product. We can do this from the *Robot Properties* dialog which can be accessed from the context menu of the robot by selecting the command with the same name. Here we set the TCP frame equal to the tools TCP frame. In Figure 4.6 is a screenshot from this dialog of a correctly configured robot.

Figure 4.6: Robot Properties



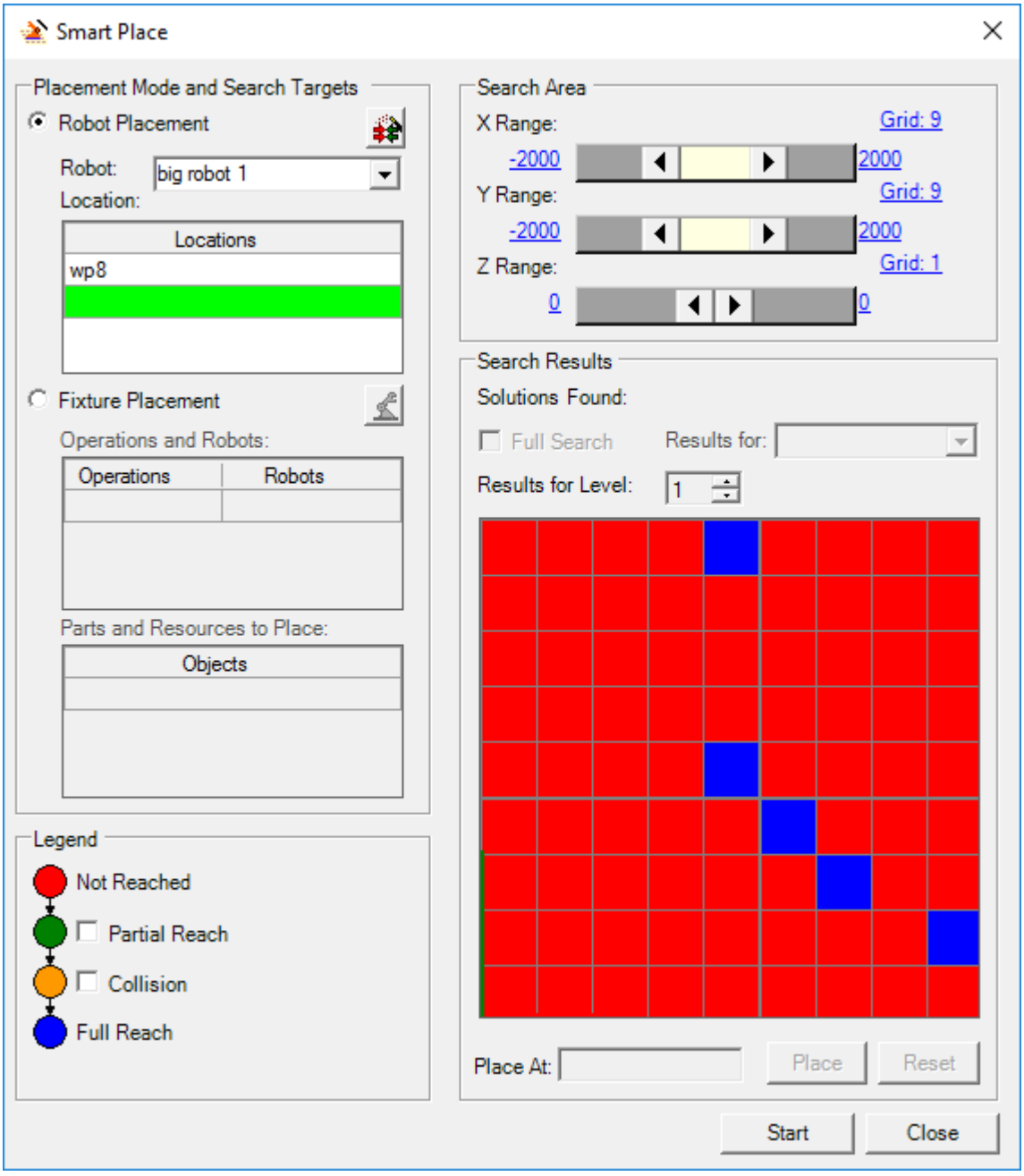
4.1.6 Positioning Robots

There is nothing special on positioning robots as far as the basics go. Robots can be placed manually using the *Placement manipulator* as any other components or resources. For robots, however, Process Simulate includes several beneficial tools. All of these tools are located as usual in the ribbon menu, in the *Robot* tab. I'd like to note a pair of them.

4.1.6.1 Smart Place

The Smart Place feature is located under *Robot >Reach >Smart Place*. It allows to quickly find places from which the specified robot will be able to reach all the specified points. The feature works by making a grid around the robot and doing a reachability test for all the points in the grid. Aforementioned grid can be seen in Figure 4.7.

Figure 4.7: Smart Place

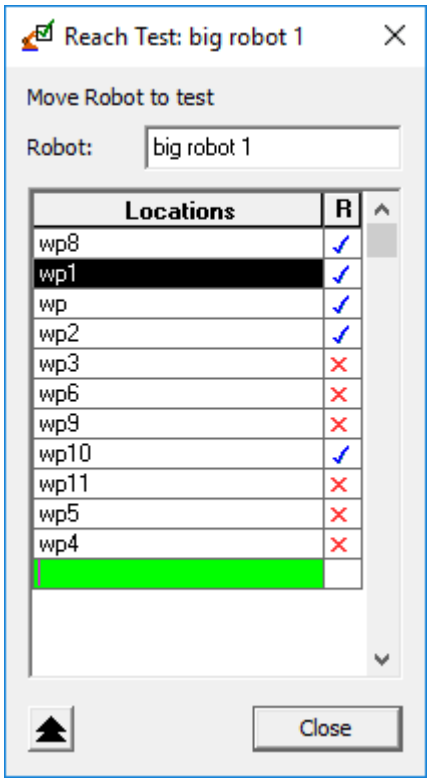


4.1.6.2 Reach test

The Reach Test feature can be found under *Robot >Reach >Reach Test*. This feature allows testing given a robot and points out which operations he

can or can't reach out of a specified list of operations. Figure 4.8 pictures the dialog controlling the feature.

Figure 4.8: Reach Test



4.1.7 Operations

Operations are a way of defining movement for the robots. The main view to interact with operations through is the *Operations Tree* panel. Operations have a root node and form a tree structure by nesting. The leaves of this tree we call points because they signify the individual locations in 3D space that the robot must visit. One layer above are operations which are linked to a robot and encompass a list of points. All the layers above are for logical grouping. Although there are many types of operations, the best example of this is a *Compound Operation* that makes up most layers above paths, including the root.

Commands for working with operations can be found on the *Operations* tab. Some common operation commands are on the *Home* tab. However, the *Operations* tab contains those and more, which are necessary when creating welding operations.

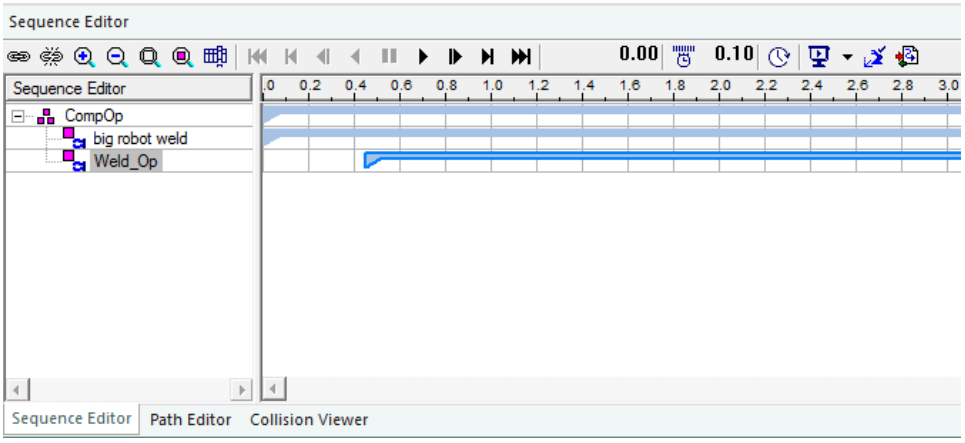
4.1.7.1 Compound Operation

Compound Operation is a very simple but an essential building block. It allows for grouping operations into a larger block and more importantly managing when and how long will each suboperation run. It enables to run multiple child operations at the same time or having one operation start after another, even composing intricate timelines.

To start managing the timeline the operation needs to be set as a current operation. You can select an operation by highlighting it in the *Operations Tree* and selecting *Set Current Operation* from the mouse context menu or using the keyboard shortcut *Shift + S*.

Process Simulate will then populate the *Sequence Editor* panel (see Figure 4.9) with information about the newly set current operation.

Figure 4.9: Sequence Editor



The order of operations within a compound operation can be reordered by dragging and dropping inside the *Operations Tree* panel. This order is merely for convenience as it doesn't affect the order of execution. For this, we have the *Sequence Editor*. In this panel, each operation is represented by a blue bar next to the operations name in the left list. These bars are draggable and represent the start, the duration and the end of the operation.

4.1.7.2 Device Operation

Device operation moves a robot to a specific pose. Which robot and into which pose should it move needs to be specified when it's being created. This operation is mainly useful for returning robots into their home position so they can be ready for the next product after finishing work on the current one.

4.1.7.3 Object Flow Operation

Object Flow Operation moves an object from one place to another. Process Simulate will present a dialog asking for a start frame, and an end frame (from and to) should the object be moved, similarly to other operations. This operation is unique by not having to be assigned to a robot.

4.1.7.4 Spot Weld Operation

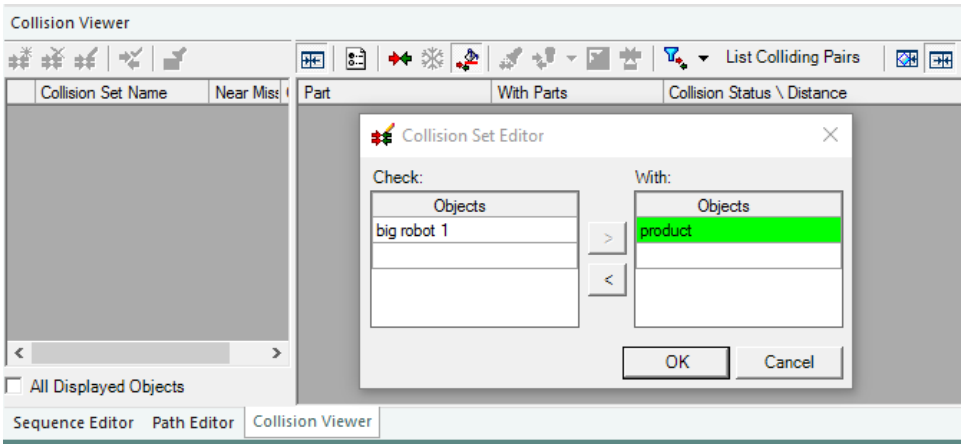
Spot welding in Process Simulate is separated into two parts: picking weld points, and combining weld points into weld operations. Spots on the product that is to be spot welded need to be designated for welding. This can be done using the *Process >Discrete >Create Weld Point by Pick/Coordinates* command. Clicking *Create Weld Point by Coordinates* will present a dialog where one can fill coordinates and select on which part the weld will happen. On the other hand *Create Weld Point by Pick* will change a cursor and allow the user to pick points in the study where to weld. These points don't yet have a part associated with them so Process Simulate can compute a perpendicular angle from the part's surface and correctly guide the robot. Unassociated points can be projected on a surface of a part using the *Project Weld Points* command in the same command group. The user will be again presented with a dialog to select all the weld points to be projected and a list of parts to project them onto.

Once each point is created and projected on a part, an operation will appear for it. This is where the second part comes in. We need to create a weld operation (*Operation >Create Operation >New Operation >New Weld Operation*), specify a robot to do the welding and populate its weld list. That is a list of the individual weld point operations we just created in the first part. The robot will then go in order of the atomic operations inside of the weld operation and process the points.

4.1.8 Detecting Collisions

Now that we defined our operations we can talk about simulating operations and detecting any problems that might occur when using that operation in the real world. We are already familiar with the *Sequence Editor* panel, of which will take further advantage. Furthermore, we'll explore the *Collision Viewer* panel which is located in the same area. Using the *Collision Viewer* panel (see Figure 4.10) we need first to define what collisions to check. Collision checking is a time-consuming process, so the fewer checks we have defined, the faster the simulation will go.

Figure 4.10: Collision Viewer



To specify a new check, we use the *New Collision Set* command which can be invoked by the first button in the *Collision Viewer* panel. You will be prompted to fill in for collisions of what objects with which objects (usually the robot and the product as depicted in Figure 4.10). Note the *Collision Options* command on the *Collision Viewer*; the dialog contains options from tolerances to stop the simulation when a collision is detected. Last but not least we need to make sure we have *Collision Mode* enabled. This option is controlled by another button in the *Collision Viewer* panel.

After we set up our collision detection, we can run the simulation. We can do so from the *Sequence Editor* by clicking the friendly looking play button. A simulation will now start moving the robots and when a collision happens a beep sound will be played unless configured otherwise.

4.2 Programming Interface

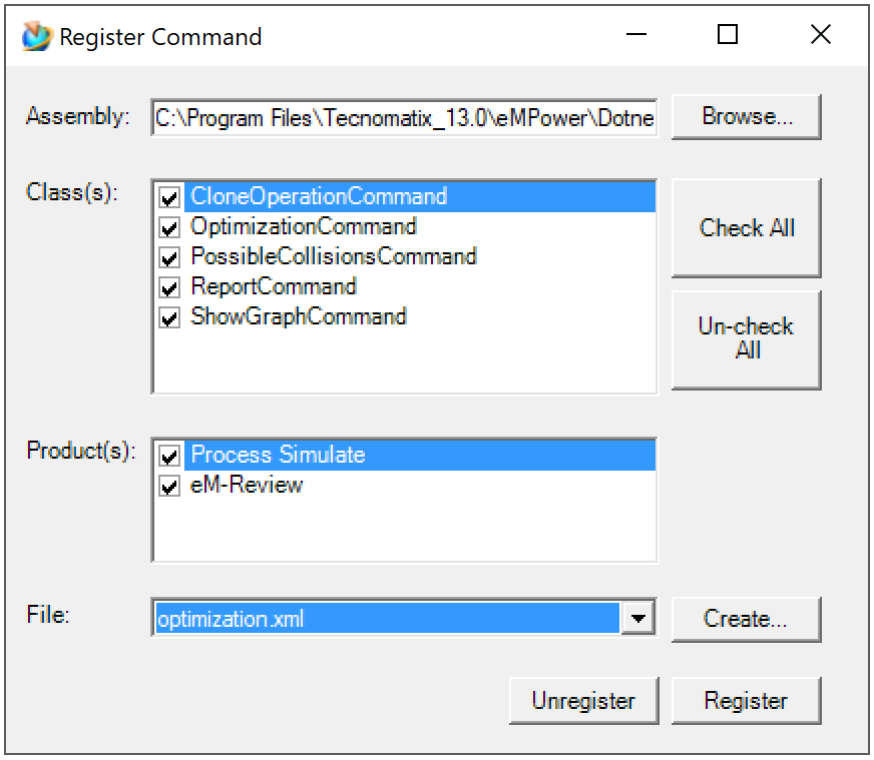
Programming for SIEMENS Tecnomatix Process Simulate was initially desirable, as it exposes a Microsoft .NET Framework compatible API. This allows developers to choose any of the many .NET languages to create Process Simulate plugins, including but not limited to C#, C++, F#, Visual Basic, Iron Python. All the code in this work is written in C#.

4.3 Writing Plug-ins

Any .NET assembly can be a Process Simulate plugin, as it only looks at the contents. A class library project is ideal for this as there is no benefit for any other project type. For Process Simulate to pick up the assembly, it should be located in *DotNetCommands* or *DotNetExternalApplications* directories and registered with the application. These directories and any following paths are relative to the programs installation directory. Typically this would be *C:/Program Files/Tecnomatix 13.0/eMPower* , but might differ based on the user's choice at install-time.

To register an external application or a command with Process Simulate, we need to use a utility *CommandReg.exe* which comes with the application. When you launch the program a dialog, similar to the one presented in Figure 4.11, will appear. In this dialog we select the compiled file, we want to load, pick the commands located in the assembly and choose a filename for the configuration XML file which will be newly created. This XML allows the settings to be moved between computers easily.

Figure 4.11: CommandReg Utility



After we registered our commands, we need to configure our workspace to be able to use them. More specifically, we need to choose where the application should display buttons for the commands in the ribbon menu. This can be achieved by right-clicking the ribbon to invoke the context menu and selecting the *Customize Ribbon* option as shown in Figure 4.12. A dialog like you see in Figure 4.13 will appear where the user can add new commands to the ribbon and customize the layout. The newly registered actions will appear in the list. Unfortunately, there is no grouping available. Therefore the best option is to look for the exact names in the alphabetically sorted list.

Figure 4.12: Customize the Ribbon

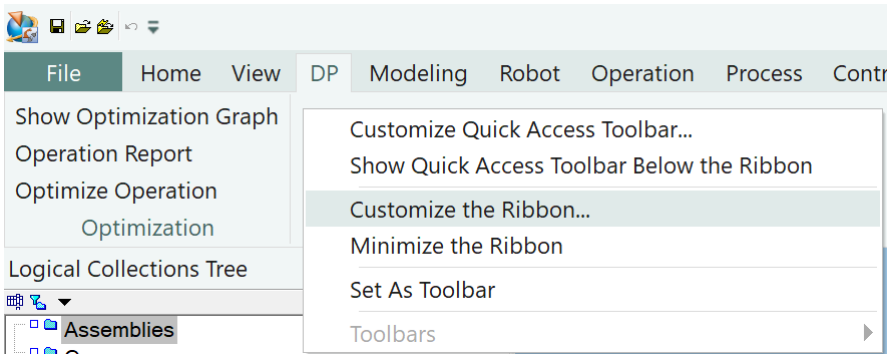
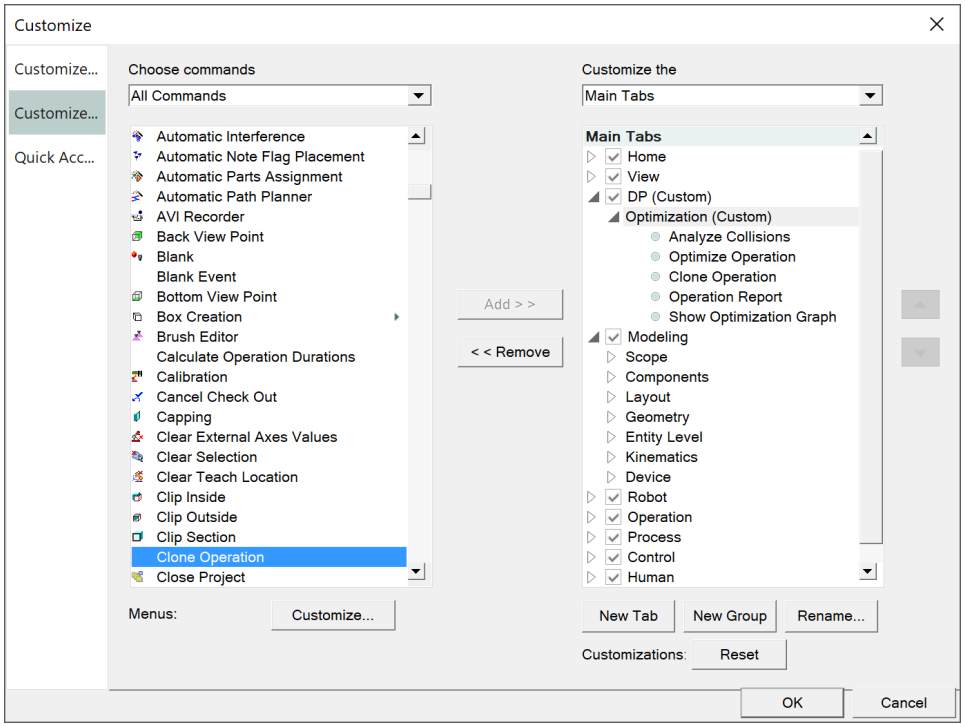


Figure 4.13: Add commands to the Ribbon



Next, we will look like at how to code new commands so that Process Simulate would recognize them. For this, we first need to include a reference for the main dynamically linked library *Tecnomatix.Engineering.dll* . Writing plugins for the application revolves mostly around this one library, as it contains all the classes used to interact with the application. There are multiple types of commands which are recognized by Process Simulate. They are named based on the UI elements they represent and range from

buttons to combo boxes. Most commonly used command type is the button, which Figure 4.14 shows implemented. To create a button command we have to create a new class and inherit the *TxButtonCommand* abstract class. After the user clicks the button, the *Execute* method will be invoked, where we put our business logic.

Figure 4.14: Example Command

```
public class MyCommand : TxButtonCommand
{
    public override String Category { get; } = "My Category";
    public override String Name { get; } = "My Command";
    public override void Execute(Object cmdParams)
    {
    }
}
```

4.4 API

This section is dedicated to acquainting the reader with the most important classes in the *Tecnomatix.Engineering.dll* library. It is not the only library available for the plugin developers, but apart from edge-cases it's the only one developers need. It allows developers to extract information from the application instance that loaded the plugin and currently the opened project. It includes routines to control some of the functions of the application.

4.4.1 TxApplication

TxApplication is a static class which serves as the main entry point to the application. This class can be readily described as the root of a tree. It holds instances of sub-services in its static properties bound to the current application instance. It also contains routines which control application specific behavior. In Figure 4.15 I picked out the most important properties and presented their signature, as I believe that for a person with development experience this is the most valuable information, and the easiest to imagine.

Figure 4.15: TxApplication API

```
public sealed class TxApplication
{
    public static TxDocument ActiveDocument { get; }
    public static TxSelection ActiveSelection { get; }
    public static TxApplicationEvents ApplicationEvents { get;
    ↪ }
    public static TxOptions Options { get; }
    public static string StatusBarMessage { set; }
    public static void RefreshDisplay()
}
```

- *ActiveDocument* is the pivotal property of this object. It contains information about the currently opened study and allows the plugin to manipulate it. The features of this object are described in Chapter 4.4.5, which talks about the *TxDocument* type.
- *ActiveSelection* allows the plugin to see what the user selected or modify the selection. This feature is especially useful when used as part of the plugins user interface, allowing the user to pick inputs for the plugin by selecting them in the application.
- *ApplicationEvents* property provides binding actions to important application events. Chapter 4.4.3 goes into more detail about how to use this property.
- *Options* contains a hierarchy of objects used to retrieve and set application options. These mimic the options found in the *Settings* dialog of the application. Chapter 4.4.4 goes into more detail.
- *StatusBarMessage* is a simple String. Modifying this property will make the text appear in the status bar area of the application. It is a simple, but effective, way to communicate with the user unobtrusively.
- *RefreshDisplay()* causes all panels in the application to redraw using the latest data.

4.4.2 TxSelection

The *TxSelection* class controls the current selection. All different kinds of items can be selected simultaneously. For this reason, the API includes this specialized class allowing the developers to interact with all kinds of selections fairly easily. In Figure 4.16 I prepared a filtered definition of the class.

Figure 4.16: TxSelection API

```
public sealed class TxSelection{
    public void Clear();
    public void AddItems(TxObjectList items);
    public void SetItems()
    public void RemoveItems()
    public ITxObject GetLastPickedItem();
    public TxTransformation GetLastPickedLocation();
    public TxObjectList GetPlanningItems();
    public TxObjectList GetAllItems();
    public TxObjectList GetOrderedItems();
    public event TxSelection_ItemsSetEventHandler ItemsSet;
    public event TxSelection_ItemsAddedEventHandler
     ↪ ItemsAdded;
    public event TxSelection_ItemsRemovedEventHandler
     ↪ ItemsRemoved;
}
```

- *GetOrderedItems()* gets the objects that are currently selected. In addition to that, this routine returns only the loaded objects, in their engineering representation in the order they were selected.

- *GetAllItems()* returns the same information, but does not guarantee order.

- *GetPlanningItem()* returns planning representations like *TxPlanningPart* or *TxPlanningResource* , in contrast to the aforementioned routines which return engineering representations like *TxRobot* or *TxComponent* .

- *GetLastPickedLocation()* returns coordinates of the last picked object.

4.4.3 TxApplicationEvents

The *TxApplicationEvents* class wraps multiple application events. The events include the closing of the application which the plugin can use to clean up temporary resources. The application exit request can also be intercepted should the user have some unsaved work in the plugin. You can see the class definition in Figure reffig:CodeTxApplicationEvents. Additionally, Figure 4.18 shows how the events in this class are used.

Figure 4.17: TxApplicationEvents API

```
public sealed class TxApplicationEvents
{
    public event TxApplication_ExitingEventHandler Exiting;
    public event TxApplication_ExitRequestEventHandler
    ↪ ExitRequest;
    public event TxApplication_ExitingEventHandler Closing;
}
```

- *Exiting* Occurs when the application is about to exit.
- *ExitRequest* Occurs before the application is about to exit. To reject the request to exit, specify *false* for the *Approve* field of the event arguments.
- *Closing* Occurs when a project is closed.

Figure 4.18: TxApplicationEvents Usage

```
class Demo()
{
    public Demo()
    {
        TxApplication.ApplicationEvents.Exiting += (sender, e)
        ↪ => {
            Save();
        };
        TxApplication.ApplicationEvents.ExitRequest +=
        ↪ (sender, e) => {
            e.Approve = !unsavedWork;
        };
    }
}
```

4.4.4 TxOptions

The *TxOptions* provides access to the application options as they mirror the options in the *Options* dialog. These options include collision checking configuration, units used, simulation and so on. Please note that spot welding options aren't available in RobotExpert, only in Process Simulate.

Figure 4.19 shows the usage of and points out the option I found the most useful. It ensures the simulation player stops playing when it reaches the first collision.

Figure 4.19: TxOptions Usage

```
TxApplication.Options.Collision.StopOnCollision = true;
```

4.4.5 TxDocument

The *TxDocument* class represents a study. When talking about the *TxApplication.ActiveDocument* object this would be the currently open study. Under the document, we can find all physical objects, operations, manufacturing features, and robotic programs. It also facilitates access to objects that have a single instance per document.

Figure 4.20: TxDocument API

```
public sealed class TxDocument
{
    public ITxOperation CurrentOperation { get; }
    public TxOperationRoot OperationRoot { get; }
    public TxPhysicalRoot PhysicalRoot { get; }
    public TxMfgRoot MfgRoot { get; }
    public TxCollisionRoot CollisionRoot { get; }
    public TxSimulationPlayer SimulationPlayer { get; }
}
```

- *OperationRoot* is the root of the operation tree
- *PhysicalRoot* is the root of the physical object tree
- *MfgRoot* is the root of the manufacturing features tree.
- *ColisionRoot* is the root of the collision pairs. It is used to determine where the workspace currently contains a collision.
- *SimulationPlayer* is used to simulate operations and events. At any given moment there is a single, current simulation player, with which all commands and viewers work.

4.4.6 Operations

Operations in Process Simulate inherit the *ITxOperation* interface. When they have children, as all operations except on the point level do, they also implement the *ITxObjectCollection* interface which serves as a nongeneric *List* specific to the Process Simulate .NET API. All operations have a name and a description.

4.4.6.1 TxOperationRoot

This class is the root of all operations. Its children are usually of type *TxCompoundOperation* . We can query all children with the *GetAllDescendants()* or *GetDirectDescendants()* methods. Operations can't be created using the new keyword invoking a constructor. To create a new operation use the *CreateXOperation()* on a class that implements *ITxOperationCreation* , like for example *TxOperationRoot* . The X in *CreateXOperation()* is the specific operation type you are trying to create, and the classes contains methods for all operation types. For example, if we wanted to create a *GenericRoboticOperation* we would use the *CreateGenericRoboticOperation* method. The created operation still needs to be inserted into a specific place in the operation tree.

Figure 4.21: TxOperationRoot API

```
bool CanCreateXOperation()
TxXOperation CreateXOperation(XCreationData creationData)
GetDirectDescendants(ITxTypeFilter filter)
GetAllDescendants(ITxTypeFilter filter)
//usage:
TxCompoundOperation newOperation
 ↪  =TxApplication.ActiveDocument.OperationRoot.CreateCompoundOperation(newTxCo
foreach (ITxOperation op
 ↪  inTxApplication.ActiveDocument.OperationRoot.GetDirectDescendants(newTxNoTy
{
}
```

4.4.6.2 TxCompoundOperation

This operation groups a set of operations (its children) into a logical group. They can also specify dependencies and offsets within the parent.

4.4.6.3 TxContinuousRoboticOperation

This path operation contains an ordered list of points. It has a robot assigned which will carry out the whole sequence of the points. In the object model, it contains a set of child links, each having a reference to a source and target operations. These links are read only but can be read and manipulated using the *GetChildAt()* and *MoveChildAfter()* functions. Timing offsets and durations are read-only which are captured after a simulation. Changing operation speeds won't recalculate this value, a new simulation needs to be performed.

4.4.6.4 TxRoboticViaLocationOperation

This operation is used to avoid obstacles since a normal operation goes for a direct approach to the target point which can result in collisions. It simply navigates the robots head to a designated point.

4.4.6.5 TxObjectFlowOperation

This operation is used for moving products from one location to another. The operation specifies how the object is supposed to be gripped with *GripFrame* (= *TxFrame*) and *GripFrameType* (= *GeometricCenter*) properties. The product will be moved through points specified by objects of type *TxObjectFlowLocationOperation* that are children of the operation (as *IEnumerable*).

Chapter 5

Integration

This chapter focuses on the plugin, how it works and how it was developed. Here I will walk the reader through the integration I implemented, its features, how I achieved it and the reasoning behind the design. First of all, I'll go through the general architecture of the solution, then explain the features this work brings and then deep dive into the essential part, the optimization process.

5.1 Architecture

The plugin is written in Microsoft.NET Framework using C# in Microsoft Visual Studio 2017. I chose this language out of all the possibilities compatible with the Tecnomatix API because I like the enterprise strategy Microsoft has taken with it. As such it pushes the users to write scalable and maintainable applications. I used the best practices in the industry to push the boundaries of readability of the code and straightforward extensibility as this was one of the goals.

The user interface of the plugin, which is just a few dialogs, is built using WPF (Windows Presentation Foundation). This technology is the state of the art for building UI on the Microsoft Windows operating system. It uses an XML based language called XAML to define the object tree of the UI elements. It supports hardware acceleration, animations, data templates and more out of the box. With each component, there isn't only a XAML file but also a so-called "code-behind" file which is standard C#. In the code-behind, we can use code to specify behavior which the XAML language can't express, or it would be too verbose.

WPF is tightly bound together with the MVVM architectural design pattern which was developed by Microsoft employees specifically for WPF. It was since recognized by the developer community as a significant pattern

in UI design and ported to many other languages and frameworks. The MVVM pattern says that Views, which we can think of as screens or dialogues and correspond to the WPF components, are controlled by so-called View Models. This way we can have multiple Views for different screen sizes or places in the application, but share the same business logic which is located in the View Model. View Models also bind the Models and the Views together. Models hold our data which the Views present to the user.

One thing the pattern doesn't talk about, but can't live without are services. Service is a part of the functionality which provides access to or implements capabilities of the application which the business logic can then use. There are multiple ways to define services, depending on the content and needs of the application. The simplest way is to define services as static classes. That is only possible if the services don't need configuration. If they do, the standard way of handling this problem is to use dependency injection to manage our services. Business logic specifies its dependencies and an IOC container will inject them. We call this principle Inversion of Control (IOC) because the business logic is no longer in control and only specifies its requirements which the container aims to satisfy. In this model, it's the responsibility of the IOC container to manage the configuration and in some cases auto-discover the services. The last recommendation is to use interfaces for services so that the business logic isn't dependant on a single implementation and the IOC container can choose between different implementations the one best suited for the situation.

While in this work I chose to go the purer route of static services as I didn't have any advanced requirements I'd like to exhibit an example of this good practice from an external code-base. Instead of using a MILP solver directly, tying the code to a specific implementation like Gurobi [14], I used an abstraction over MILP solvers in general. Because of this I could use an open-source solver like LP_SOLVE [15] or Google OrTools [16] and have confidence that when a more complicated MILP model comes, the solver can be quickly swapped for a more sophisticated solver like Gurobi [14] or CPLEX [17].

5.2 Commands

In the previous chapter, I described what a command is, how to develop them and how to add them to the ribbon menu. Installing this plugin into Process Simulate will allow the user to add commands contained in this assembly onto his command bar. The following commands are a part of the plugin.

- *Clone Operation.* After selecting an existing operation and setting it

as active, invoking this command will create a deep copy. The same function will be used for optimization backup. Therefore it's an excellent way to test if the backup is going to meet quality standards. It can also be used to create clones for experimentation without disturbing the primary workspace. This capability was implemented using the internal API used for the drag & drop functionality of Process Simulate.

- *Optimize Operation.* This command is the entry point to the main functionality of the plugin. It displays a dialog to configure and start the optimization process. The process is described in detail in a forthcoming section.

- *Analyze Collisions.* This command will run a simulation which analyzes collisions each time-frame. The output of this process is a list of operations that caused collisions which will be displayed on the screen. In contrast to the functionality included in Process Simulate that only provides the user with information about colliding objects.

- *Operation Report.* This command will run a simulation which analyzes the active operation and its descendants and produces read-outs of the data. First, a report outlining the energy usage of every robot at every time-frame. Second, a report exposing joint speeds and accelerations of the different robots at every time-frame and finally a report showing the current robot settings. Then the user will be offered to pick file locations to where *csv* files with these three datasets should be created.

- *Show Optimization Graph.* This command strives to demonstrate what the optimization graph looks. It uses the *GraphVisualizationService* class to construct a model of the operation identical to the one used by the optimization process. It serializes this instance to a Graphviz format and uses *dot.exe* to draw it to an image. Graphviz [18] needs to be installed on the local machine and the *dot.exe* executable must be in the path for the image to get produced.

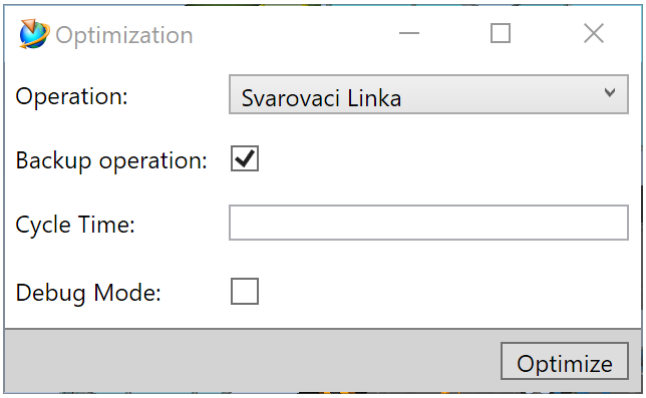
- *Interpolate Speed.* This command uses the operation *OperationDurationInterpolator* class to adjust the speed of the operation to match the specified duration. A point operation must be active, and the duration must lie within the maximum and minimum duration attainable.

5.3 Optimization Process

The optimization process consists of the main optimization routine and several modules that provide the required functionality for the algorithm which

then glues it together. This chapter will first explain the optimization process and then focus on the supporting modules in the sub-sections.

Figure 5.1: Optimization Dialog



The user first triggers the optimization dialog (see Figure 5.1) by clicking on the Optimize button provided by the plugin. The dialog allows the user to adjust parameters of the optimization process. First, the system presents him with the operation which should be optimized, which is the operation he had currently set the as the active operation so that he has a chance to adjust this before starting the optimization process. In the following text, I will refer to this operation as the root operation. The user is also presented with the choice to optionally backup the operation which will create a copy of the root operation and then set the duplicate as the new root. Setting a specific cycle time in seconds will make the MILP model try to match the cycle time instead of minimizing it. Lastly, the debug option can provide the user with additional information from the MILP solver, explaining the solution, should he have troubles with the proposed schedule.

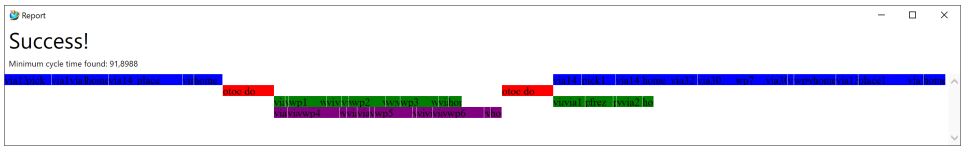
The dialog uses the optimization service which is implemented in the *OptimizationService* class located in the *Tecnomatix.Optimization.Services.Optimization* namespace. This service is responsible for the main optimization flow described below. The service has one dependency, and that's an optimization provider (*IOptimizationProvider*). In this thesis I included an optimization provider which uses the MILP model defined in Chapter 3 which optimizes cycle time. It is implemented in the *DefaultOptimizationProvider* class and the service mentioned above defaults to this provider. It can also be swapped out for a different provider which implements the *IOptimizationProvider* interface for example for a different criteria function.

When started, the optimization process will first aim to gather data about the root operation. It will first simulate the behavior of the root operation while the robots are set to maximum and minimum speeds to obtain the minimum and maximum duration of the individual child activities.

Then an optimization graph is built by analyzing the operation tree. This graph is also augmented with the information about the simulation and any initial collisions the process might have encountered. After this initial batch of data is gathered the optimization cycle can start. In this loop, the graph is given to a solver which proposes a solution. The process adjusts the root operation to match the proposed solution, and the result is simulated. If there are no collisions, we have the best viable solution, and the cycle ends. Otherwise, we advance to the next round. To prevent an infinite loop, this can only be repeated a limited amount of times. This constant is defined in the *OptimizationService* class.

When the process finishes the system will display a helpful report (see Figure 5.2) presenting the most important information about the solution to the user. In the header the cycle time can be found and the body is filled with a graphical representation of the schedule.

Figure 5.2: Optimization Result Dialog



5.3.1 Simulation

The simulation module is a wrapper around the kinematic simulation capability of Process Simulate. It is meant to help with obtaining data about the study which can only be measured. The simulation services are placed in the *Tecnomatix.Optimization.Services.Simulation* and revolve around the *Simulation* class. All new simulations must derive from this class and then can be easily executed. The engine also supports executing multiple simulations simultaneously.

To run a simulation, the system needs an operation as an input. Process Simulate will block the UI and compute the movement of the robots in the scope of the given operation, frame by frame. While simulating the player fires events, which the specific simulation can choose to subscribe to if they are relevant for its purpose. Usually, as the playback progresses, the

simulation will store some data which when processed will form the output values. Events available range from the simulation being started or ending, operations beginning and ending to the individual time intervals at which the locations of all the objects within the study are computed.

A report is a specialized type of simulation. This simulation gathers data while the simulation is in progress and processes them. This data can then be exported into a *csv* file. Reports are useful for debugging or gaining insight into the performance of the activities within the study.

The plugin comes with the following simulations implemented:

- *Collision Simulation.* This simulation is checking every frame if there is a collision between the specified collision pairs. The collision pairs can be adjusted in the *Collision Viewer* panel in Process Simulate. The simulation will also track down the responsible operations which controlled the parts that collided. This is done by assigning to each operation a set of objects which can be moved by the operation and intersecting them with the set of objects that collided. It is assumed that a collision can happen only between two operation controlled objects, otherwise this is a fault of the robotic cell design which optimization can't fix and the collision is ignored.

- *Energy Report.* This report captures the simulated energy usage of the robots. Please note that the readings are only as accurate as the robot controller and using a dedicated controller from the manufacturer of the robots is recommended as the readings given by the default robot controller are inaccurate.

- *Joint Speed Report.* This report captures the speeds of the joints of the robots while performing the various tasks engulfed in the robotic operation.

- *Duration Simulation.* To capture how long it will take for a robot to perform the operation it is first needed for a simulation to run. This information will be captured by Process Simulate afterward, but if any changes happened to the study, it might be inaccurate. For this reason, to obtain accurate readings, this simulation will capture the durations right after the simulation finishes. It is also able to adjust the speed of the robots before running the simulation and restore the original values when done.

- *Operation Speed Report.* This report extracts the speeds the robots are set to run at while executing the particular operations.

5.3.2 Graph

The heart of the optimization process is a model of the problem. We defined this model in Chapter 2 and the optimization graph is just a set of entities capturing it in code. The *OptimizationGraph* class represents the graph as a whole. The vertices and edges also have their classes, those are *OptimizationVertex* and *OptimizationEdge* respectively. Properties of all of the classes correspond to what we defined in Chapter 2.

On top of holding data, the *OptimizationGraph* class also contains methods for serialization (JSON) and visualization of the graph. Out of the box, there are two visualization options available.

- Graph View. Presented in Figure 2.1, this view is the natural view of the graph. It shows all the edges, vertices, and values of the major properties. This view can be generated by the *ExportDiagram* method and uses the *QuickGraph* library for Microsoft .NET to serialize the nodes and relations into a *Graphviz* [18] *dot* format. The Graphviz library, specifically the *dot.exe* executable is used to turn the graph description into a picture. I chose the Sugiyama-style [19] hierarchical layout, implemented in the Graphviz library, due to the good quality of the results.
- Schedule View. Presented in Figure 5.2, this view focuses on the proposed solution. It visualizes the times and durations of the individual operations. In this view, the y-axis shows the different processing units (robots) and the x-axis plots time. It can be generated by the *ExportTimeline* method and the diagram is constructed in the HTML format, each operation corresponding to a *div* element with absolute positioning which is calculated based on the proposed duration, proposed start, and the index of the robot associated with the operation. It can be displayed in a dialog by an embedded web browser.

5.3.3 Graph Builder

Putting together the optimization graph which serves as the model of the study is one of the most fundamental aspects of the optimization process. It takes a compound operation as an input and traverses all of its descendants transforming it into a flat structure. It also has to analyze the dependencies of the operations and how would it operate in a production cycle.

This functionality is being provided by the *GraphVisualizationService* which is located in the *Tecnomatix.Optimization.Services* namespace. It can generate a graph either at bottom-most level of points or one level higher, at the level of paths. For the optimization process used in the point

level graph we use the point level graph.

First, the optimization graph is filled with vertexes which are the children of the root operation lying at the desired level, grouped by the robot which is assigned to perform them. In each group, the operations are ordered and a so-called "robot loop" is created, by linking the operations together in pairs, each one with the next. One exception to this rule is the edge between the first and the last operation, which are linked by a "robot loop reset" edge instead.

The hardest part of the graph creation are the "link" edges which can appear at any level in the operation tree. Since we are flattening the tree into a graph, we need to project them into this new structure. This process is different whether we flatten to the level of paths or points.

Let's say that $descendants(x)$ is a function which returns all descendants on the level of paths (for example on Figure 5.3, if Operation 1 was given as an input to this function, output of the function would be only the children in the frame "Paths"). Next we need to define the function $first(x)$ which returns the first child (a point) of path x and $last(x)$ returns the last child of x.

In the first case, it is possible to just find all the descendants at the level of paths of the source of the link $l = (source, target)$ and of the target of the link. Then we compute the Cartesian product of these two sets $S = descendants(source), T = descendants(target), P = S \times T$. For each item in the set $\forall(from, to) \in P$ we create a new link of type "link" which leads from the vertex *source* to the vertex *target* in the graph. These nodes are guaranteed to exist if the link leads inside the root operation. If it leads outside its is ignored because it wouldn't have any impact on the run of the operation.

In the latter case, when creating a graph at the level of points, the situation is similar, however, after forming the Cartesian product, we have to select the correct points from the paths. A custom recursive function *ProjectOntoGraph* achieves this. The projection which is a result of the function is illustrated in Figure 5.3. This function first creates the Cartesian product $P = S \times T$ on the path level, as in the previous case. Then it selects the last or first point $\forall(from, to) \in P : edge = (last(from), first(to))$, depending if the set is coming from a source or a target operation respectively. And then create the corresponding *edge* in the graph. For example on Figure 5.3 this function would project the blue operation link into the nine purple edges shown in the middle of the diagram.

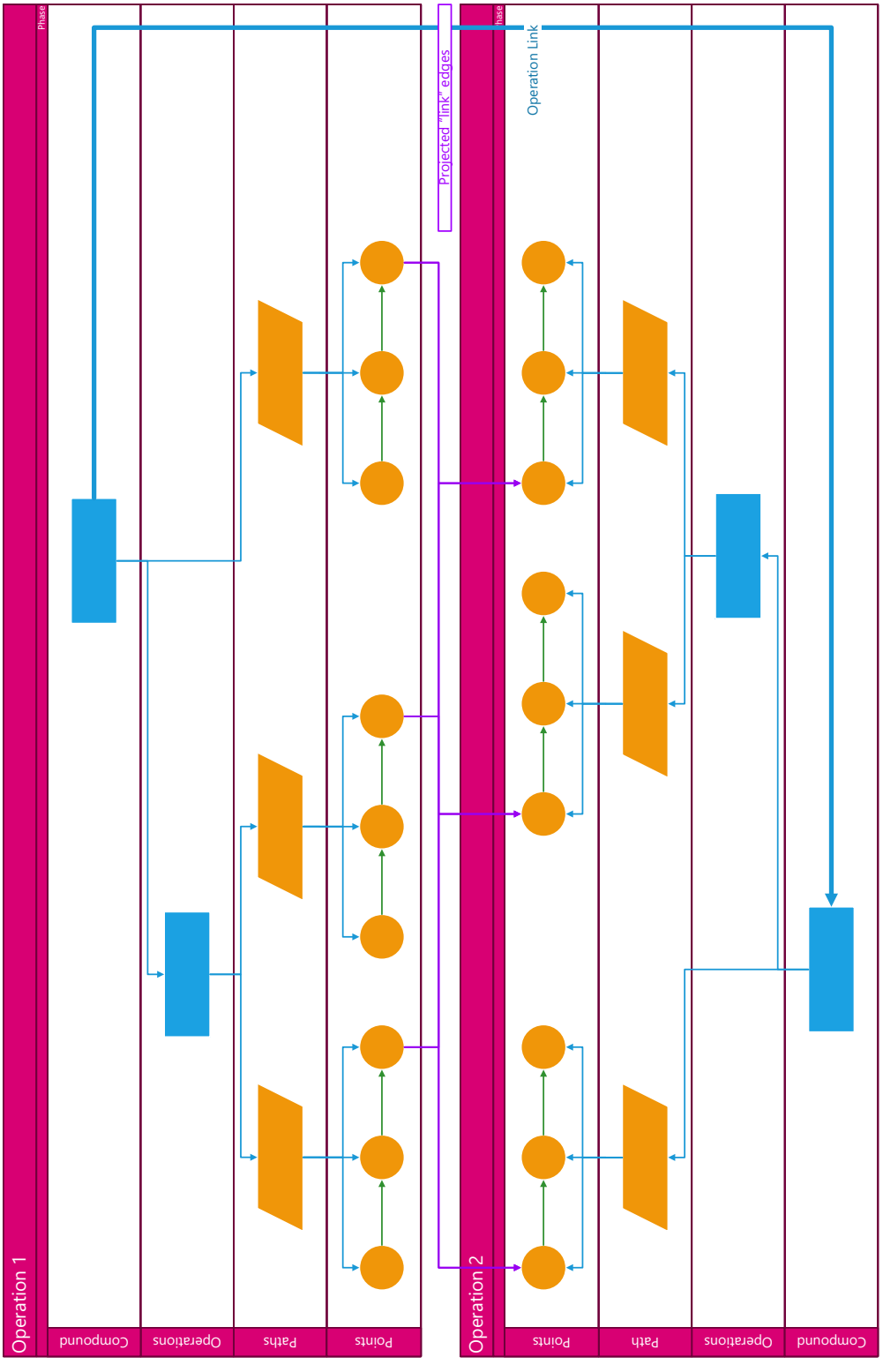
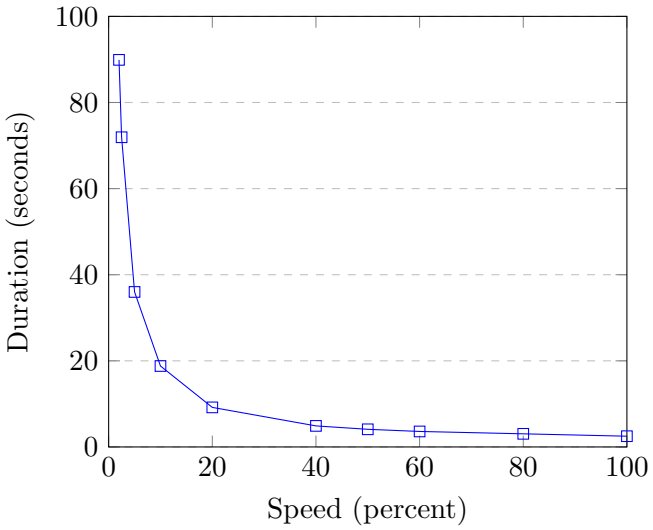


Figure 5.3: Projection of the operation tree onto a flat layer. An operation link between two compound operations (blue) is projected into 9 projected links between points (purple).

5.3.4 Interpolator

The interpolator is the main ingredient in the process responsible for taking the output of the optimization and adjusting the operation accordingly. Since how the component works and how its used is very closely tied together, I will explain both in this section. But before we jump to the algorithms we need to realize a few unintuitive facts which we determined experimentally.

Figure 5.4: Relation of operation duration to robot speed



Firstly, as you can see in Figure 5.4, the relation between duration, which the optimization algorithm calculates, and speed which we need to set for the robots, isn't linear. Unfortunately, the curve also varies from robot to robot and from operation to operation which means the value can't be calculated at all. This challenge we tackle by using a modified interpolation search with a simulation to check what is the actual duration.

The second challenge stems from the fact that movement operations can have different precision associated with them. For example, when moving around a corner when the precision requirements are minimal, we can set the operation zone as "coarse" which will allow the robot to move to the next operation after reaching a region around the point specified by a configured tolerance. As opposed to a "fine" zone which will require the robot to move precisely to the specified point and only then it can advance to the next operation. Due to this, the speed of an operation can influence the following operation, and some operations are effectively skipped which implies that adjusting their speed will not change the duration. To address this, I

added an upper bound to the MILP model for the operation duration. An operation which is being skipped will then have $\underline{d} = d = \overline{d}$, consequently the optimization engine won't try to adjust this operations duration and therefore its speed.

To use the interpolator to adjust speeds of an operation we first need to instantiate the *OperationDurationInterpolator* class located in the *Tecnomatix.Optimization.Services.Interpolation* namespace. First, we need to set the desired processing durations of the points which can be achieved by the *SetBlockDuration* method. Please note that the interpolator is designed to adjust multiple operations at the same time because the process needs simulations to check that the adjusted speed resulted in the correct duration. Since simulations are by far the most expensive process, it is a good practice to use the least amount of simulations as possible. The interpolator can also adjust operations in blocks in case it didn't make sense to adjust the operation individually, but only the cumulative duration was important. In this case, all the operations in the block will be adjusted to the same speed, and the sum of their durations will be matched to the specified desired duration. After the interpolator has all the data, we can begin the calibration process by calling the *Adjust* method on the instance of the interpolator class. This is a blocking operation due to the fact that the simulations are blocking the UI thread.

First of all the interpolator holds a lower and upper bound for the speed and duration of each operation. These bounds are set to a speed of 1 (minimum) and the speed of 100 (maximum) and the durations observed for those speeds. Similarly to an interpolation search, the interpolator will iterate until it finds an acceptable solution or decides that it is impossible or impractical to continue. In the loop, the first action is to make a prediction of the speed based on the upper and lower bounds and the desired duration. At first, I considered linear interpolation (Equation 5.1), but because the relation isn't linear as shown on Figure 5.4, binary halving (Equation 5.2) was used instead because it should converge in fewer iterations in most cases.

$$prediction = S_{min} + \frac{D}{D_{max} - D_{min}} \times (S_{max} - S_{min}) \tag{5.1}$$

$$prediction = \frac{S_{min} + S_{max}}{2} \tag{5.2}$$

After the algorithm makes predictions for the whole batch, the next step is to modify the joint speed of all the operations in the blocks based on the projections and test them with a simulation. Based on the results of

the simulation, either the upper or the lower bound of each block is modified.

Some readers, acquainted with Process Simulate, might be wondering why we don't convert the movement kind of the operation from joint speed to motion type and write the proposed duration in there. The reason is that the motion type duration only affects the period of the movement, and any OLP commands or welding action take extra time on top of that. Ideally, if we could find out the length of the static part, we could subtract it from the total duration, and still write out the times as motion type. Alas, there is no way to determine the length of the two distinct parts, and therefore we can't use that feature and have to discover the joint speed by simulation.

5.3.5 Generator

The generator piece does as the name suggests. It allows for the programmatic generation of random optimization graphs which can be used for automated testing. These graphs are not guaranteed to be feasible.

Random graphs can be generated using the *GraphGenerationService* located in the *Tecnomatix.Optimization.Services.Generation* namespace.

The inputs to the routine are three parameters which control the size and density of the graph. The variables are *robots* which is the number of robot loops, *operationsPerRobot* which specifies the number of operations in each loop $\pm variance$. After this trivial graph is created, the process aims to expand the model to mirror a real-life scenario. A number of random dependencies between robots are inserted (equal to *operationsPerRobot*), and the same amount of collisions are artificially constructed to complicate the graph.

Ordinarily, the graph vertices are linked to Tecnomatix representations of the operations and robots so this time they are replaced by virtual operations and robots so that most processes can function with this graph.

Chapter 6

Experiments

Due nature of the problem not all of the testing and validation could be performed automatically and compared with related works. For this reason, I split the formal validation of this work into two parts. First I assess the performance of the MILP model and determine whether a heuristic needs to be used instead. Then I perform manual testing evaluating the quality of the optimization process on predefined test scenarios.

6.1 Performance Testing

To benchmark the performance of the provided optimization provider, which consists of a MILP model and a selected MILP solver for the model, I chose to feed it randomly generated models and measure the time it took the provider to come up with a solution. The randomly generated models were created by the generator module which is explained in detail in Chapter 5.3.5. I used this module to generate 100 random models for each complexity setting and then averaged the results into the number which you can see in the Table 6.1. A complexity setting of n corresponds to a robotic cell with n co-operating robots. Each robot has $n+2$ actions to act out, and the overall system includes n collisions. To ensure the instances aren't trivial also n inter-robot dependencies are added. I tested complexity settings 1 to 40 as the higher complexity settings took too much time to process 100 times and having 40 robots is not practical in a single robotic cell.

I run the benchmarks on a computer equipped with Intel Xeon E3-1230v2 @ 3.30 GHz [20] and 32 GB of RAM. The architecture of the plugin allows for a smooth replacement of the MILP solver. However, the presented benchmarks were performed using the open source engine LP_SOLVE [15].

Based on the results shown in Figure 6.1 obtained I decided that a heuristic solution wasn't needed for this model as even in the most extreme case of

40 robots the solver takes only 22 seconds. On the other hand, the architecture of the plugin doesn't bind the optimization providers to a MILP solver, and for more complex models a heuristic solution can be used. Considering the duration of the simulations in the optimization process, 22 seconds for a run of the optimization provider is an acceptable value.

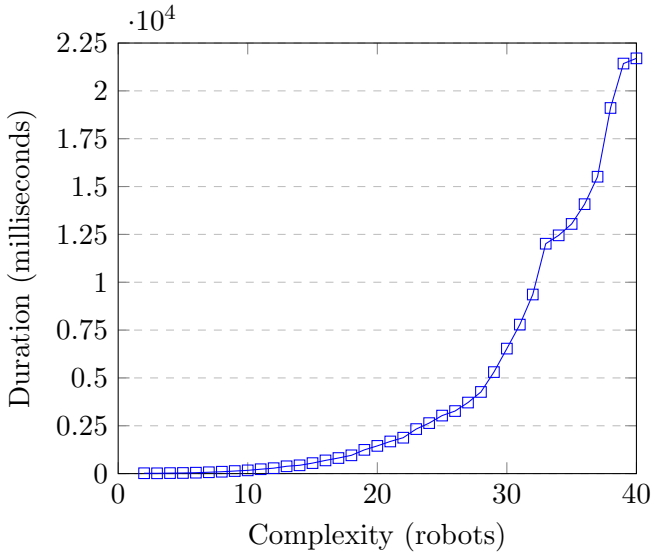


Figure 6.1: Growing duration with complexity of the model

6.2 User Testing

Unit testing is preferable when it comes to assuring the quality of components of an application. However, components that interact with Process Simulate need a study to be loaded. That, unfortunately, requires user input and can't be done programmatically because of it, therefore can't be unit tested. In this chapter, I present the experiments that were carried out for the components that require a study and the optimization process overall.

The testing was performed two different studies. A group of engineers from a professional environment provided the first, realistic, study seen on Figure 1.1 ("weld test study"). And the second was a custom study created for testing collisions ("collision test study"). Unfortunately, both contain proprietary information and can't be released as a part of this thesis.

The weld test study simulates a robotic cell, where the product comes in a box. The central robot picks up the robot from the box and places it on a turning table. The table turns and allows access to the product to two welding robots which perform some welding on the side of the product.

Then the table turns back, and the central robot picks up the product again. This time it puts it into a stationary punch, holding it in place, while the punch makes a hole. Meanwhile one of the welding robots puts his tool into a grinding device which cleans up the residue from welding. After the hole is punched, the central robot transfers the product into another stationary machine and then back into the box. Multiple processes are executed at parallel showcasing all types of links and operations in the study.

The collision test study consists of two robots facing each other, one moving his tool frame on a line ("line operation"), while the other is periodically crossing the line ("cross operation"). Each of the paths is made out of four operations. Depending on the operation speed the robots would either collide or not. Of course, collision detection is enabled and configured with appropriate collision pairs.

6.2.1 Interpolator

The Interpolator component was manually tested according to the following scenario:

- Open Process Simulate
- Load the weld test study
- Select a point operation and mark it as active
- Invoke the *Interpolate Speed* command and enter a value between the maximum and minimum
- Write down the resulting operation speed and compare to expectations

A point operation was selected from the test study. When running it at minimum speed (10%) a maximum duration of 3.1s was obtained and likewise at maximum speed (100%) a minimum duration of 0.34s was observed. I selected the duration of 2s as the target and executed the command. When the process has finished I observed a duration of 2.00s and the interpolated speed of 16.42 in the *Path Editor* window.

6.2.2 Graph Builder

The Graph Builder component was manually tested according to the following scenario:

- Open Process Simulate
- Load the weld test study

- Select a compound operation and mark it as active
- Invoke the *Show Optimization Graph* command
- Compare the presented diagram to the operation tree

When testing the graph builder, I selected the top-most compound operation representing the whole welding process of using multiple robots cooperating. After executing the command, I compared each node and the values within. More importantly, I made a point of checking all the edges to match operation tree.

6.2.3 Collision Analysis

The Collision Analysis component was manually tested according to the following scenario:

- Open Process Simulate
- Load the collision test study
- Select the compound operation and mark it as active
- Invoke the *Analyze Collisions* command
- Compare the result with the expected outcome (1 collision)
- Adjust the operation speed of the line operation to 50%
- Invoke the *Analyze Collisions* command
- Compare the result with the expected outcome (0 collisions)

The *Analyze Collisions* command outputs a list of operations that were being executed at the time of the collision and had a relationship with one of the objects causing the collision. It is therefore important to also check the operation names to match the expectation.

6.2.4 Operation Backup

The Operation Backup component was manually tested according to the following scenario:

- Open Process Simulate
- Load the weld test study
- Select any operation and mark it as active

- Invoke the *Clone Operation* command
- Compare the operation trees by simulation
- Compare the links to resources and appearances

After testing with the users, I found out that even though the operation tree copy is perfect, specific bindings, namely appearances, are not always preserved. That is something that I can't add to the process because there is no API for manipulating appearances. I sincerely hope the Process Simulate team will fix this in a future version of the application since it also affects other capabilities using this API like drag & drop or copy & paste.

6.2.5 Optimization Process

The optimization process was tested on both studies to highlight specific aspects of the optimization algorithm. The testing process included the following scenarios.

- Open Process Simulate
- Load the collision test study
- Select the root operation and mark it as active
- Invoke the *Optimize Operation* command and click "Optimize"
- Assess the quality of the solution (a valid solution without collisions)

In the collision test the algorithm found a solution, avoiding possible collisions but increasing the root operation duration. This is not the minimum possible cycle time, but the minimum cycle time that could have been found with the "safe" collision resolution method used.

- Open Process Simulate
- Load the weld test study
- Select the root operation and mark it as active
- Find an operation on the critical path and decrease its speed to 50%
- Invoke the *Optimize Operation* command and click "Optimize"
- Assess the quality of the solution (all operations on the critical path have speed set to 100%)

The final test for the plugin will be a trial run in a professional environment, which will determine the future direction of this project.

Complexity	Constraints	Variables	Time
2	235	231	20ms
3	332	335	17ms
4	449	464	27ms
5	566	593	33ms
6	708	752	48ms
7	892	963	71ms
8	1081	1179	98ms
9	1298	1431	134ms
10	1477	1635	170ms
11	1750	1954	229ms
12	1950	2184	284ms
13	2318	2621	383ms
14	2498	2825	434ms
15	2849	3241	549ms
16	3224	3684	690ms
17	3522	4034	814ms
18	3812	4374	956ms
19	4349	5017	1241ms
20	4730	5468	1448ms
21	5153	5971	1680ms
22	5422	6286	1876ms
23	6054	7044	2327ms
24	6475	7544	2639ms
25	6995	8166	3032ms
26	7261	8476	3269ms
27	7711	9012	3713ms
28	8228	9629	4267ms
29	9100	10681	5305ms
30	9662	11354	6532ms
31	10156	11943	7787ms
32	10550	12410	9359ms
33	11578	13652	12013ms
34	11743	13841	12451ms
35	12574	14842	13046ms
36	12960	15299	14080ms
37	13810	16323	15514ms
38	14865	17599	19106ms
39	15670	18569	21427ms
40	16136	19124	21700ms

Table 6.1: MILP Solver Benchmark

Chapter 7

Conclusion

The main goal of my thesis was to explore how we can increase the effectivity of manufacturing systems, by integrating a CAD software designed for simulating manufacturing processes with an optimization algorithm.

First, I had to familiarize myself with the Process Simulate application and the manufacturing domain. I examined the features of the application and created a simple assembly line consisting of three robots working on a product that moved on a conveyor belt.

Then I had to investigate options for creating plugins for the software. This part could have been much easier if not for the fact that documentation for the Tecnomatix suite is practically non-existent. I created a guide on how to create new commands that will appear in the ribbon bar of the application. I also set up a development environment where the build process will automatically produce a library which the application can read and start it so that it could see the results quickly.

I explored the functionality officially available to developers and documented them. For specific capabilities, like reading out the energy usage of robots, I was able to find an undocumented way to retrieve this information. Based on the available functionality, which was unfortunately quite limiting, I designed a process that analyzes the open study and uses several different algorithms determine the optimal operation settings. This process can use an optimization module which defines the criteria for optimality. I devised a MILP model that optimizes the cycle time of an operation and transformed it into code. Together these functions form a helper the users can use to make sure their robotic cells produce products at maximum efficiency.

To test the correctness of the plugin, and its performance, I created a generator of artificial models with variable complexity. I used the genera-

tor to generate a hundred models for each complexity level and allow the algorithm to solve them. The performance of the solver was measured and analyzed. I found the performance of the solver satisfying, especially compared to the time spent by simulation, and therefore a heuristic algorithm was not implemented.

The plugin was also tested manually with practical examples of robotic cells provided by engineers experienced in the field.

7.1 Future work

Unfortunately, I can't claim to have solved the topic of optimization in manufacturing or saved billions of any currency. A lot more work has to be done on this application for it to be usable in a professional environment. As it usually is, progress, especially in science, is incremental. As I was building on the shoulders of giants, I'd like to propose few areas where to take this work in the future.

- *Energy Consumption Optimization.* Even within optimal cycle time, we can look at sleep modes and operation speeds to reduce energy usage. Some preliminary work was already made [4] with good results.
- *More accurate predictions for the interpolation search algorithm.* Predicting better speeds based the duration could result in fewer cycles and therefore fewer simulations which take most of the time in the duration of the optimization process.
- *Specialized behavior for different scenarios.* Different kinds of operations have to be treated in their specific way. Handling as many kinds as possible could significantly improve the flexibility of the algorithm and also its results.
- *More granular collision handling.* Currently, If there is a collision between two operations the operations are forbidden to be operating simultaneously at any stage of the execution. A more precise algorithm could track down the problematic section of the operation make sure only that portion is restricted so that the operations can still overlap, albeit partially.

Bibliography

[1] Chi G. Lee and Sang C. Park. "Survey on the virtual commissioning of manufacturing systems". In: *Journal of Computational Design and Engineering* 1.3 (2014), pp. 213–222. ISSN: 2288-4300. DOI: https://doi.org/10.7315/JCDE.2014.021. URL: http://www.sciencedirect.com/science/article/pii/S2288430014500292.

[2] Siemens Product Lifecycle Management Software. *Customer Case Studies by Company*. 2017. URL: https://www.plm.automation.siemens.com/en/about_us/success/customer-case-studies/.

[3] Ray Y. Zhong, Xun Xu, Eberhard Klotz, et al. "Intelligent Manufacturing in the Context of Industry 4.0: A Review". In: *Engineering* 3.5 (2017), pp. 616–630. ISSN: 2095-8099. DOI: https://doi.org/10.1016/J.ENG.2017.05.015. URL: https://www.sciencedirect.com/science/article/pii/S2095809917307130.

[4] L. Bukata, P. Šůcha, Z. Hanzálek, et al. "Energy Optimization of Robotic Cells". In: *IEEE Transactions on Industrial Informatics* 13.1 (Feb. 2017), pp. 92–102. ISSN: 1551-3203. DOI: 10.1109/TII.2016.2626472.

[5] GM Media. *GM Commits to 100 Percent Renewable Energy by 2050*. 2016. URL: http://media.gm.com/media/us/en/gm/home.detail.html/content/Pages/news/us/en/2016/sep/0914-renewable-energy.html.

[6] D. Meike and L. Ribickis. "Energy efficient use of robotics in the automobile industry". In: *2011 15th International Conference on Advanced Robotics (ICAR)*. June 2011, pp. 507–511. DOI: 10.1109/ICAR.2011.6088567.

[7] European Commision. *Energy Efficiency*. 2017. URL: https://ec.europa.eu/energy/en/topics/energy-efficiency.

[8] R.G. Fenton, D. Poon, and S.P. Davies. "Robotic Workcell Cycle Time Optimization Using Computer Graphics". In: *CIRP Annals* 41.1 (1992), pp. 509–512. ISSN: 0007-8506. DOI: https://doi.org/10.1016/S0007-8506(07)61256-6. URL: http://www.sciencedirect.com/science/article/pii/S0007850607612566.

[9] J. Zhang and X. Fang. "Robot move scheduling optimization for maximizing cell throughput with constraints in real-life engineering". In: *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Dec. 2013, pp. 221–227. DOI: 10.1109/ROBIO.2013.6739462.

[10] E. Åblad, D. Spensieri, R. Bohlin, et al. "Intersection-Free Geometrical Partitioning of Multirobot Stations for Cycle Time Optimization". In: *IEEE Transactions on Automation Science and Engineering* PP.99 (2017), pp. 1–10. ISSN: 1545-5955. DOI: 10.1109/TASE.2017.2761180.

[11] Anne-Laure Coiffier. "Analysis and design of manufacturing operations". PhD thesis. Czech Technical University in Prague, Mar. 2017.

[12] Claire Hanen and Alix Munier. "A study of the cyclic scheduling problem on parallel processors". In: *Discrete Applied Mathematics* 57.2 (1995). Combinatorial optimization 1992, pp. 167–192. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(94)00102-J. URL: http://www.sciencedirect.com/science/article/p%20ii/0166218X9400102J.

[13] Siemens AG. *Process Simulate*. 2017. URL: https://www.plm.automation.siemens.com/en/products/tecnomatix/manufacturing-simulation/assembly/process-simulate.shtml.

[14] Gurobi. *Gurobi Optimizer*. 2017. URL: http://www.gurobi.com/products/gurobi-optimizer.

[15] Peter Notebaert Kjell Eikland. *lpsolve*. 2017. URL: http://lpsolve.sourceforge.net.

[16] Google Inc. *Google Optimization Tools*. 2017. URL: https://developers.google.com/optimization/.

[17] IBM Corporation. *CPLEX Optimizer*. 2017. URL: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer.

[18] *Graphviz - Graph Visualization Software*. 2017. URL: https://graphviz.gitlab.io/.

[19] K. Sugiyama, S. Tagawa, and M. Toda. "Methods for Visual Understanding of Hierarchical System Structures". In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (Feb. 1981), pp. 109–125. ISSN: 0018-9472. DOI: 10.1109/TSMC.1981.4308636.

[20] *Intel® Xeon® Processor E3-1230 v2*. 2017. URL: https://ark.intel.com/products/65732/Intel-Xeon-Processor-E3-1230-v2-8M-Cache-3_30-GHz.

Appendix A

Abbreviations

.NET Framework for writing applications developed by Microsoft

CAD Computer Assisted Design

API Application Programming Interface

MVVM Model–View–ViewModel architectural pattern

WPF Windows Presentation Foundation

XAML Extensible Application Markup Language

DI Dependency Injection

IOC Inversion of Control

MILP Mixed-Integer Linear Programming

APPENDIX A. ABBREVIATIONS

Appendix B

CD Contents

- **/bin** - Folder containing the compiled plugin
- **/src**
  - **/Tecnomatix.Optimization.sln** - The Visual Studio Solution
  - **/Tests** - The unit test project
  - **/Tecnomatix.Optimization** - The plugin project
- **/doc**
  - **/thesis.zip** - source files of the text part of this thesis (LaTeX)
  - **/thesis.pdf** - this text in PDF
  - **/research/apiresearch.md** - documentation for Process Simulate API with code examples